**POLITECHNIKA CZĘSTOCHOWSKA**
**WYDZIAŁ INŻYNIERII MECHANICZNEJ I INFORMATYKI**
**INSTYTUT MATEMATYKI I INFORMATYKI**

Praca magisterska

# ARCHITEKTURA I IMPLEMENTACJA SILNIKA GRAFIKI TRÓJWYMIAROWEJ

ARCHITECTURE AND IMPLEMENTATION OF 3D GRAPHICS ENGINE

**Adam Sawicki**
kierunek: Informatyka
specjalność: Inżynieria oprogramowania i systemy informatyczne
nr indeksu: 41344

Promotor
dr inż. Mariusz Ciesielski

CZĘSTOCHOWA 2008

# Spis treści

**1. Wprowadzenie** **5**
1.1. Przegląd literatury . . . . . . . . . . 7
1.2. Charakterystyka dostępnych bibliotek graficznych . . . . . . . . . . 19
1.3. Cel i zakres pracy . . . . . . . . . . 21

**2. Architektura silnika** **23**
2.1. Podstawowe założenia . . . . . . . . . . 24
2.2. Biblioteki zewnętrzne . . . . . . . . . . 26
2.3. Architektura biblioteki bazowej . . . . . . . . . . 26
2.4. Architektura szkieletu . . . . . . . . . . 51
2.5. Architektura silnika . . . . . . . . . . 71
2.6. Narzędzia i eksportery . . . . . . . . . . 87

**3. Implementacja silnika** **91**
3.1. Proces renderowania . . . . . . . . . . 91
3.2. Implementacja shadera głównego . . . . . . . . . . 99
3.3. Format pliku modeli 3D . . . . . . . . . . 112
3.4. Implementacja animacji szkieletowej . . . . . . . . . . 115
3.5. Implementacja efektów cząsteczkowych . . . . . . . . . . 117
3.6. Implementacja efektów postprocessingu . . . . . . . . . . 121
3.7. Renderowanie terenu . . . . . . . . . . 126
3.8. Efekt opadów atmosferycznych . . . . . . . . . . 130
3.9. Renderowanie trawy . . . . . . . . . . 131
3.10. Renderowanie nieba . . . . . . . . . . 134
3.11. Renderowanie drzew . . . . . . . . . . 136
3.12. Renderowanie wody . . . . . . . . . . 137
3.13. Implementacja swobodnego drzewa ósemkowego . . . . . . . . . . 139
3.14. Budowa mapy . . . . . . . . . . 144

**4. Przykłady** **147**
4.1. Przykłady gier komputerowych . . . . . . . . . . 147
4.2. Przykłady efektów graficznych . . . . . . . . . . 150

**5. Podsumowanie** **157**

**Bibliografia** **161**

# Rozdział 1

# Wprowadzenie

Dziedziną, której dotyczy ta praca i zarazem szczególnym obszarem zainteresowań autora jest **renderowanie akcelerowanej sprzętowo, trójwymiarowej grafiki komputerowej czasu rzeczywistego**. Aby uściślić to określenie, warto opisać każdą jego część:

- **Grafika komputerowa** [1] to dziedzina informatyki związana z operowaniem na obrazach.
- **Grafika trójwymiarowa** to taka, w której obiekty reprezentowane są w przestrzeni 3D i dopiero na koniec odzwierciedlane na płaszczyźnie ekranu.
- **Renderowanie** to generowanie (wytwarzanie) obrazu na podstawie danych wejściowych, w odróżnieniu od przetwarzania istniejącego obrazu czy też jego analizowania.
- **Grafika czasu rzeczywistego** wymaga, aby obraz powstawał nie dłużej niż około $0.03$ sekundy, co pozwala na pokazywanie przynajmniej 30 klatek obrazu na sekundę i tym samym daje złudzenie płynnego ruchu oraz zapewnia interaktywność.
- **Grafika akcelerowana sprzętowo** to taka, w której do renderowania obrazu i uzyskania wydajności wymaganej do działania w czasie rzeczywistym wykorzystywane jest wsparcie nowoczesnych kart graficznych.

**Grafika komputerowa** posiada liczne zastosowania, pośród których wymienić moż na projektowanie (np. inżynierskie, architektoniczne), wizualizację (np. naukową, biznesową), zastosowania artystyczne oraz rozrywkowe (np. filmy, gry).

**Grafika czasu rzeczywistego** stanowi z punktu widzenia programisty szczególne wyzwanie. W grafice takiej nie można sobie pozwolić na wyliczanie obrazu dowolnymi metodami zapewniającymi pożądaną jakość, tak jak to jest robione w przypadku przygotowania współczesnych filmów animowanych. Konieczność zapewnienia dostatecznie krótkiego czasu powstawania obrazu wymusza zaawansowane optymalizacje, jak najlepsze wykorzystanie dostępnego sprzętu oraz dobór stosowanych technik i algorytmów, które zapewnią odpowiednią wydajność przy jak najmniejszej utracie jakości.

Do zastosowań grafiki komputerowej czasu rzeczywistego należą m.in. symulacje (służące np. do szkolenia pilotów czy żołnierzy) oraz gry komputerowe. **Gry komputerowe**, jakkolwiek służą celom rozrywkowym, muszą być przy tym brane pod uwagę jako bardzo poważne zastosowanie, ponieważ przemysł gier rozwija się obecnie na świecie bardzo szybko i osiąga obroty porównywalne z przemysłem filmowym. Terminu „gry komputerowe" (albo też „gry wideo") nie należy mylić z dziedziną matematyki nazywaną teorią gier, gdyż są to zupełnie różne pojęcia.

**Tworzenie gier** jako proces biznesowy składa się z wielu elementów i wymaga współpracy specjalistów wielu dziedzin takich jak programiści, graficy, muzycy, projektanci, testerzy, i inni. Powstanie gry komputerowej wysokiej jakości (tzw. tytuły „AAA") oznacza lata pracy oraz kosztuje nierzadko miliony dolarów.

Z punktu widzenia programisty, **programowanie gier** to specyficzna dziedzina, która łączy w jednym wspólnym celu m.in. inżynierię oprogramowania, sztuczną inteligencję, symulacje fizyczne, przetwarzanie sygnałów oraz właśnie renderowanie akcelerowanej sprzętowo, trójwymiarowej grafiki czasu rzeczywistego. Tematem niniejszej pracy jest wyłącznie ta ostatnia. Programowanie gier odróżnia od programowania innego rodzaju aplikacji wiele cech. Przykładowo, ze względu na wydajność działania kodu, językiem programowania obowiązującym w tej dziedzinie jest C++ [2], podczas gdy aplikacje biznesowe najczęściej pisze się w językach łatwiejszych do opanowania i użycia, takich jak Java albo C#.

Wraz z rozwojem gier komputerowych i wzrostem ich złożoności powstał odrębny element składowy tego typu aplikacji zwany silnikiem. Silniki bywają tworzone i licencjonowane jako osobne produkty i coraz więcej firm decyduje się na tworzenie gier w oparciu o zewnętrzny silnik zamiast tworzyć własny. Silnik nie jest pojęciem ściśle zdefiniowanym. Intuicyjnie można powiedzieć, że **silnik** (w informatyce) to rozbudowana biblioteka programowa realizująca zasadniczą funkcjonalność aplikacji danego rodzaju. Mianem silnika przyjęło się przy tym określać pewne rodzaje bibliotek takie jak: silnik renderujący w przeglądarce internetowej WWW czy silnik bazy danych, a w przypadku gier komputerowych — silnik graficzny, silnik fizyczny oraz silnik gry.

**Silnik graficzny** albo renderer to biblioteka realizująca renderowanie grafiki czasu rzeczywistego. W celu realizacji tego zadania wykorzystuje ona bibliotekę graficzną pozwalającą na użycie akceleracji sprzętowej oferowanej przez kartę graficzną. Obecnie na platformie PC bibliotekami takimi są DirectX [3] oraz OpenGL [4]. Silnik graficzny jest potrzebny w bardziej złożonych grach, ponieważ stanowi dodatkową warstwę abstrakcji. Korzystając bezpośrednio z biblioteki graficznej, programista musi zajmować się renderowaniem trójkątów, manipulowaniem teksturami, buforami, shaderami i innymi zasobami niskiego poziomu. Silnik daje tymczasem dostęp do obiektów bardziej abstrakcyjnych, jak scena, encja, światło czy materiał, ukrywając przed użytkownikiem szczegóły implementacyjne.

## 1.1. Przegląd literatury

Dostępne jest dużo literatury poświęconej grafice komputerowej czasu rzeczywistego, ale źródła bywają na różnym poziomie zaawansowania — od tzw. internetowych *tutoriali* adresowanych do początkujących, poprzez książki kompleksowo wprowadzające do tematyki programowania gier, książki bardziej wyspecjalizowane i adresowane do bardziej zaawansowanych programistów, aż po artykuły i prezentacje naukowe, często pochodzące z konferencji takich jak SIGGRAPH czy Game Developers Conference. Literatury w języku angielskim jest dostępnej wielokrotnie więcej, niż w języku polskim.

Pozycje wprowadzające w tematykę programowania grafiki czy też uczące korzystania z biblioteki graficznej takie jak DirectX z przyczyn oczywistych nie są tutaj opisane. Bardzo ważnym źródłem w tym kontekście jest oficjalna dokumentacja dostarczona przez producenta danej biblioteki (np. **DirectX SDK**). Na wzmiankę zasługuje natomiast literatura wprowadzająca podstawy teoretyczne grafiki komputerowej, jak [1].

Istotnym zagadnieniem jest **wiedza matematyczna** potrzebna do implementacji silnika. Jej zakres obejmuje przede wszystkim geometrię analityczną z pojęciami specyficznymi dla tej dziedziny, jak macierze transformacji i kwaterniony w zastosowaniu do reprezentowania rotacji i orientacji. Istnieje kilka pozycji książkowych wykładających podstawy matematyczne potrzebne w programowaniu grafiki i gier. Spośród nich autor poleca [5].

Książek poświęconych typowo architekturze silnika graficznego czy silnika gry jest stosunkowo niewiele [6, 7]. Wiele zagadnień potrzebnych podczas implementacji silnika zostało zawarte w [8].

**Podział przestrzeni** Kluczowym elementem optymalizacji podczas renderowania grafiki czasu rzeczywistego jest jak najszybsze (jak najprostsze i jak najwcześniejsze) odrzucanie możliwie dużych elementów z dalszego renderowania w danej klatce. Owo odrzucanie, nazywane też przycinaniem (ang. *culling*) odbywa się na wszelkich etapach procesu renderowania. Na poziomie procesora karty graficznej (GPU) jest to odrzucanie pojedynczych pikseli (*Z–Test*, *Alpha–Test*, *Stencil–Test*) oraz trójkątów (*Backface Culling*). Na poziomie procesora głównego (CPU) może to być odrzucanie całych obiektów, co do których można łatwo stwierdzić, że są poza zasięgiem pola widzenia (ang. *Frustum Culling*).

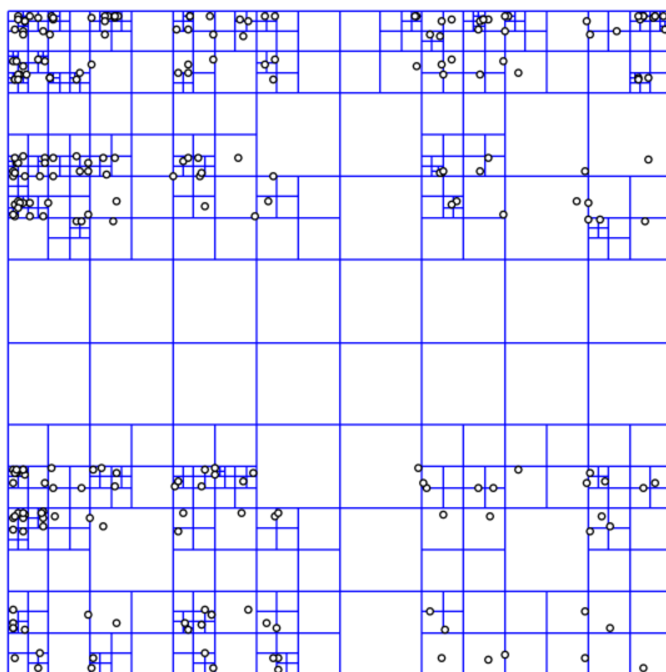
Na jeszcze wyższym poziomie można odrzucać całe fragmenty wirtualnego świata (sceny). Potrzebny jest do tego pewien podział sceny na części i jej reprezentacja za pomocą specjalnej struktury danych. Wynaleziono wiele takich struktur, a używające ich techniki nazywane są technikami podziału przestrzeni.

Trzeba przy tym podkreślić, że sceny trójwymiarowe dzieli się ogólnie na przestrzenie zamknięte (ang. *Indoor*) i przestrzenie otwarte (ang. *Outdoor*). **Przestrzenie**

**zamknięte** składają się z układu pomieszczeń połączonych korytarzami. **Przestrzenie otwarte** natomiast pokazują rozległy teren z ustawionymi na nim obiektami (jak drzewa, budynki) oraz sklepieniem niebieskim. Inne techniki podziału przestrzeni są skuteczne w zastosowaniu do przestrzeni zamkniętych, a inne w zastosowaniu do przestrzeni otwartych. Osobnym zagadnieniem jest skuteczne połączenie tych dwóch rodzajów scen tak, aby wirtualna kamera mogą między nimi przechodzić w sposób płynny (ang. *Seamless*).

Niektóre najpopularniejsze techniki podziału przestrzeni to:

- **BSP** (ang. *Binary Space Partitioning*) polega na podziale przestrzeni dowolnie zorientowanymi płaszczyznami, co w efekcie pozwala uzyskać drzewo binarne. Historię wynalezienia tej techniki podaje [9].
- **Drzewo K–D** (ang. *kd–tree*) polega na podziale przestrzeni płaszczyznami zawsze prostopadłymi do osi układu współrzędnych.
- **Drzewo czwórkowe** (ang. *Quadtree*) polega na rekurencyjnym podziale prostokątnego obszaru na cztery prostokątne podobszary. Ta struktura danych została opisana w [10]. Zobacz rys. 1.1
- **Drzewo ósemkowe** (ang. *Octree)* polega na rekurencyjnym podziale prostopadłościanu na osiem prostopadłościanów podrzędnych. Tworzy drzewo, w którym każdy węzeł jest albo liściem, albo posiada 8 podwęzłów.
- **Portale** to technika polegająca na podziale mapy na sektory połączone tzw. portalami. Obszar widoczny w zasięgu kamery jest przycinany przez portale, co pozwala stwierdzić, które sektory są widoczne z danego punktu widzenia.



Rys. 1.1. Drzewo czwórkowe dzielące rekurencyjnie płaszczyznę pozwala szybko wyszukiwać rozmieszczone na niej punkty. (Źródło: Wikipedia)

Techniki podziału przestrzeni to bardziej ogólne pomysły niż konkretne algorytmy. Można je modyfikować, a nawet łączyć. Istnieje wiele często używanych modyfikacji, na przykład swobodne drzewo ósemkowe (ang. *Loose Octree*) opisane w [11].

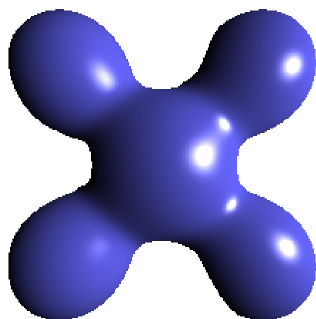
**Oświetlenie i materiały** Zagadnieniem kluczowym w renderowaniu realistycznej grafiki są obliczenia oświetlenia. Na wysokim poziomie abstrakcji łączą się w tym miejscu parametry opisujące materiał, z którego wykonana jest dana powierzchnia oraz światło, które na nią pada. Na niskim poziomie celem jest wyliczenie ostatecznego koloru piksela. Oświetlenie polega przy tym tak naprawdę na przyciemnianiu (ang. *shading*) oryginalnego koloru materiału.

Metody oświetlenia w grafice czasu rzeczywistego podzielić można na statyczne i dynamiczne. **Oświetlenie statyczne** to takie, dla którego jasność poszczególnych miejsc na scenie (np. na podłodze, suficie, na ścianach) została wcześniej obliczona i jest tylko prezentowana. To pozwala na obliczenie jej dokładnymi, dającymi realistyczne efekty metodami z grupy oświetlenia globalnego (ang. *Global Illumination*), które są zbyt wolne do zastosowania w czasie rzeczywistym (np. metoda energetyczna — ang. *Radiosity* albo metoda map fotonowych — ang. *Photon Mapping*). Wadą tego podejścia jest niemożność zmiany warunków oświetleniowych w czasie pracy programu. Wyliczony kolor i jasność oświetlenia w poszczególnych miejscach składowana jest zwykle na tzw. mapach światła (ang. *Lightmap*), choć istnieją też bardziej zaawansowane rozwiązania, np. *Radiosity Normal Mapping* [12].

**Oświetlenie dynamiczne** polega na wyliczaniu wpływu światła na każdy punkt powierzchni w czasie rzeczywistym. To podejście dzieli się dalej na oświetlenie ***per vertex***, polegające na dokonywaniu obliczeń tylko dla wierzchołków siatki i interpolowaniu wyników na powierzchni trójkątów (co odpowiada obecnie obliczeniom w Vertex Shaderze) oraz ***per pixel***, polegające na obliczaniu pełnego oświetlenia dla każdego piksela rysowanego obrazu (co odpowiada obecnie obliczeniom w Pixel Shaderze). Oświetlenie *per vertex* jest — jak łatwo można się domyślić — bardziej wydajne, ale zarazem wyglądające dużo mniej realistycznie. Popularne obecnie na rynku karty graficzne posiadają już wydajność i możliwości pozwalające na realizowanie dynamicznego oświetlenia *per pixel*.

Podstawą obliczeń oświetlenia jest **cieniowanie Gourauda** [13]. Zobacz rys. 1.2. Do tych obliczeń potrzebne są przede wszystkim wektor wskazujący kierunek od danego punktu do źródła światła oraz wektor normalny, prostopadły do powierzchni w tym punkcie. Ponieważ w grafice 3D czasu rzeczywistego powierzchnie reprezentowane są za pomocą siatki trójkątów, wektory normalne zapisywane są w wierzchołkach i interpolowane na powierzchni trójkątów, co pozwala uzyskać w miarę gładkie cieniowanie nawet dla słabo steselowanych powierzchni.

Głównym rodzajem oświetlenia jest oświetlenie rozproszone (ang. ***Diffuse***) wyliczane zgodnie z prawem Lamberta. Prawo to mówi, że intensywność światła padają-



Rys. 1.2. Obiekt z widocznym oświetleniem typu rozproszonego (ang. *Diffuse*) i odblaskiem (ang. *Specular*). (Źródło: Wikipedia)

cego na powierzchnię jest wprost proporcjonalna do cosinusa kąta między kierunkiem wskazującym na źródło światła, a kierunkiem prostopadłym do powierzchni. Ten z kolei jest równy iloczynowi skalarnemu wspomnianych wyżej dwóch wektorów (o ile są one znormalizowane), co pozwala szybko obliczać takie równanie.

Drugim ważnym składnikiem oświetlenia, obecnym na powierzchniach błyszczących, jest odblask (ang. ***Specular***). Do jego obliczenia potrzebna jest znajomość trzeciego wektora wskazującego kierunek od danego punktu do pozycji obserwatora (kamery). Istnieją dwa najważniejsze modele odblasku. Model Phonga [14] jest dokładniejszy, natomiast model Blinna [15] jest prostszy do obliczania.

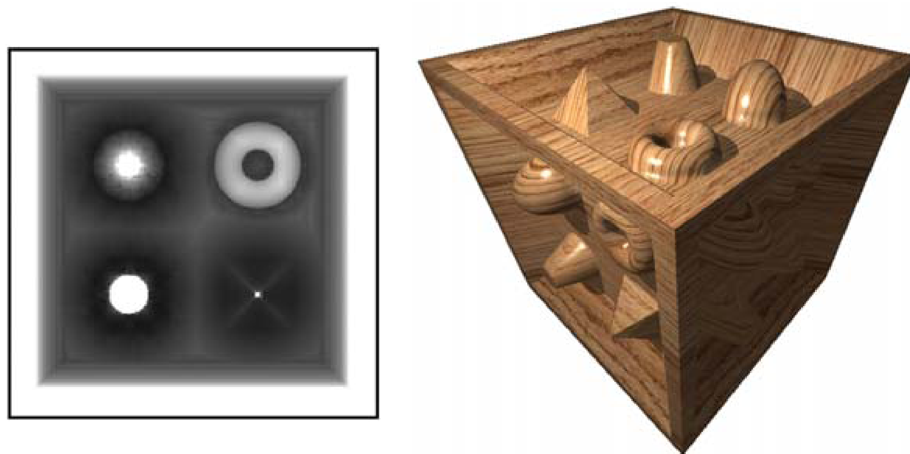
Wynalezione zostało wiele modeli efektów, których zastosowanie może dodatkowo wzbogacić efektowność i realizm renderowanych scen. Alfa–blending [16] (ang. ***Alpha–Blending***) pozwala na uzyskiwanie obiektów półprzezroczystych. Mapowanie środowiskowe (ang. ***Environmental Mapping***) [17] pozwala uzyskać odbicie obrazu sceny otaczającej obiekt na tym obiekcie. ***Half–Lambert*** [18] to dodatkowa modyfikacja wzoru Lamberta przydatna w niektórych sytuacjach. Oświetlenie anizotropowe (ang. ***Anisotropic Lighting***) [19, 20] to rzadko stosowany efekt pozwalający na oświetlenie powierzchni posiadającej mikrostrukturę, takiej jak włosy, tkanina, polerowany metal.

**Mapowanie tekstur i wypukłości** Trójkąty, z których złożona jest cała renderowana grafika, nie są pokrywane jednolitym kolorem. Nawet w dobie kart graficznych zdolnych narysować w czasie rzeczywistym wiele milionów trójkątów na sekundę, drobne szczegóły powierzchni nie są reprezentowane przez gęstą siatkę trójkątów, ale przez nierzadko całkiem duże trójkąty pokryte rozciągniętym na ich powierzchni dwuwymiarowym, bitmapowym obrazem, który w tym kontekście nazywany jest teksturą.

Mapowanie tekstur (ang. ***Texture Mapping***) wynalazł Edwin Catmull [21]. Od tego czasu powstało — i nadal powstaje — wiele technik bardziej zaawansowanego mapowania tekstur, które nie tylko odwzorowują kolorystykę, ale także fakturę nierówności materiału, z którego wykonany ma być wirtualny obiekt.

Najprostszą techniką mapowania nierówności (ang. *Bump Mapping*) jest mapowanie normalnych (ang. ***Normal Mapping***) [22]. Polega ona na używaniu do obliczeń oświetlenia *per pixel* wektorów normalnych pobranych ze specjalnej tekstury zwanej mapą normalnych (ang. *Normal Map*), w której w składowych kolorów RGB (czerwony, zielony, niebieski) upakowane są tak naprawdę składowe $(x, y, z)$ tych wektorów. Zastosowanie mapy normalnych wyrażonej w przestrzeni stycznej (ang. *Tangen Space*) wymaga przekształcania do tej przestrzeni pozostałych wektorów biorących udział w obliczaniu oświetlenia (jak wektor kierunku do światła czy kierunku do obserwatora). Wektory tworzące bazę tego układu współrzędnych to, oprócz wektora normalnego (ang. *Normal*), dwa wektory styczne do powierzchni w danym punkcie, nazywane po ang. *Tangent* i *Binormal*.

Dalsze polepszenie wizualnej jakości renderowanej powierzchni można uzyskać stosując mapowanie uwzględniające różnice w przesunięciu punktów znajdujących się wyżej względem tych bardziej zagłębionych w wirtualnej fakturze powierzchni, czyli efekt paralaksy. Służy do tego ***Parallax Mapping*** [23]. Oryginalny efekt posiada jednak wady, dlatego powstało wiele jego odmian, jak *Relief Mapping* [24], *Parallax Occlusion Mapping* [25], *Cone Step Mapping* [26], czy opublikowany w ubiegłym roku *Relaxed Cone Stepping* [27]. Zobacz rys. 1.3.



Rys. 1.3. *Relaxed Cone Stepping* (po prawej) wykonany na sześcianie z użyciem specjalnie przygotowanej tekstury (po lewej). (Źródło: [27])

**Renderowanie cieni** W naturze cień powstaje tam, gdzie nie dochodzi światło. W grafice komputerowej cień może powstawać w ten sam sposób podczas obliczania oświetlenia metodami z grupy *Global Illumination*. Nie są to jednak metody nadające się do zastosowania w czasie rzeczywistym przy mocy obliczeniowej, jaką dysponuje współczesny sprzęt graficzny. Z kolei w technikach oświetlenia stosowanych obecnie w grafice czasu rzeczywistego cień nie powstaje automatycznie. Dlatego renderowanie cieni jest osobnym zagadnieniem.

Istnieją dwie główne techniki stosowane do renderowania możliwie dokładnego,

rzucanego przez obiekty cienia w czasie rzeczywistym. Bryła cienia (ang. ***Shadow Volume***) [28] wykorzystuje posiadany przez karty graficzne bufor szablonu (ang. *Stencil Buffer*) i rysuje do niego geometrię obiektu rzucającego cień, rozciągniętą w kierunku padania światła. Przecięcia tej geometrii bryły cienia z normalną geometrią sceny pozwalają na wyliczenie miejsc, w których dany obiekt zasłania źródło światła. Zaletą tej metody jest dokładne odwzorowanie kształtu obiektu. Wadą natomiast — duże zapotrzebowanie na wydajność karty graficznej w kwestii renderowania pikseli (ang. *Fillrate*), trudność w uzyskaniu miękkich krawędzi cienia oraz konieczność specjalnego przygotowania geometrii obiektu do jej rozciągania w kierunku padania światła.

Drugą stosowaną powszechnie techniką jest mapowanie cienia (ang. ***Shadow Mapping*** — SM) [29]. W tworzonych i wydawanych obecnie grach i silnikach graficznych zdaje się ona uzyskiwać pozycję dominującą. Opiera się na dodatkowym przebiegu renderującym całą scenę z punktu widzenia źródła światła do specjalnej tekstury zwanej mapą cienia (ang. *Shadow Map*), w której zamiast koloru zapisana jest głębokość (czyli jakby odległość najbliższego punktu od światła w danym kierunku). Następnie ta tekstura jest wykorzystywana przy normalnym renderowaniu sceny oświetlonej danym światłem w celu stwierdzenia, czy dane miejsce rysowanej powierzchni nie jest zasłonięte przez jakiś punkt leżący bliżej źródła światła w linii prostej.

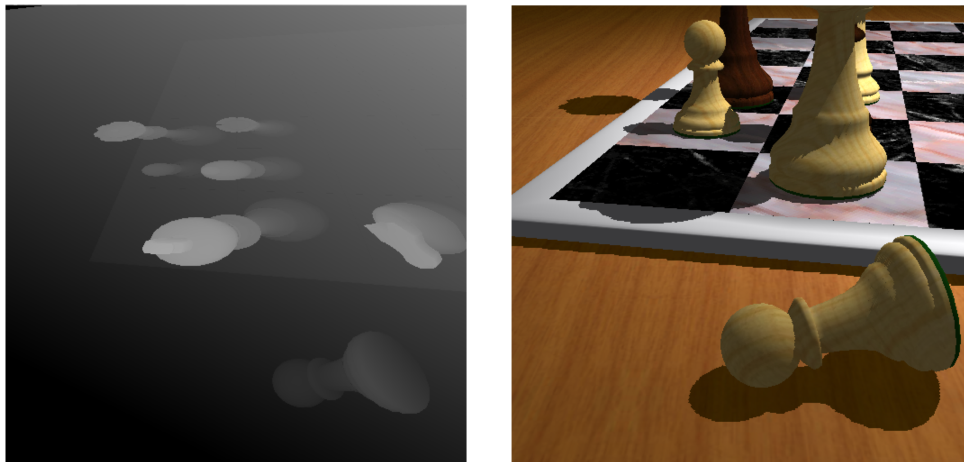
Technika mapy cienia wymaga odpowiednich operacji matematycznych (przekształceń układu współrzędnych) oraz wykonywania testu porównującego odległość danego punktu z odległością zapisaną w mapie cienia. Powstaje przy tym wiele problemów, które liczni badacze próbowali rozwiązać na różne sposoby. Podstawowym kryterium użyteczności danej metody jest możliwość jej realizacji z użyciem akceleracji sprzętowej na współczesnych kartach graficznych.

Jednym z problemów są „ząbkowane" krawędzie cienia wynikające ze skończonej rozdzielczości mapy cienia. W celu wygładzenia tych krawędzi zaproponowane zostało filtrowanie tekstury, z których najprostszym jest ***Percentage Closer Filtering*** (PCF) [30]. Daje ono przy okazji złudzenie miękkiego cienia, które jednak dalekie jest od w pełni realistycznego. Dlatego powstało wiele sposobów na jego rozwinięcie. Karty graficzne z układem firmy nVidia potrafią wykonywać PCF sprzętowo [31].

Innym problemem jest rzucanie cienia przez światło typu punktowego (ang. *Point Light*), które świeci z jednego punktu we wszystkich kierunkach (jak np. żarówka). Mapa cienia dla takiego światła musi je otaczać ze wszystkich stron. Najprostszym rozwiązaniem jest użycie w tym celu tekstury sześciennej (ang. *Cube Texture*), jednak to wymaga osobnego renderowania całej sceny do każdej z 6 ścian tej tekstury. Zaproponowano optymalizację polegającą na użyciu tylko dwóch tekstur i mapowania dwuparaboloidalnego (ang. ***Dual Paraboloid Mapping***) [32]. Jednak z eksperymentów autora niniejszej pracy wynika, że dla bardziej złożonych scen, w przeciwieństwie do prostych scen testowych, technika ta jest niemożliwa do zastosowania z powodu

nieliniowego charakteru transformacji paraboloidalnej.

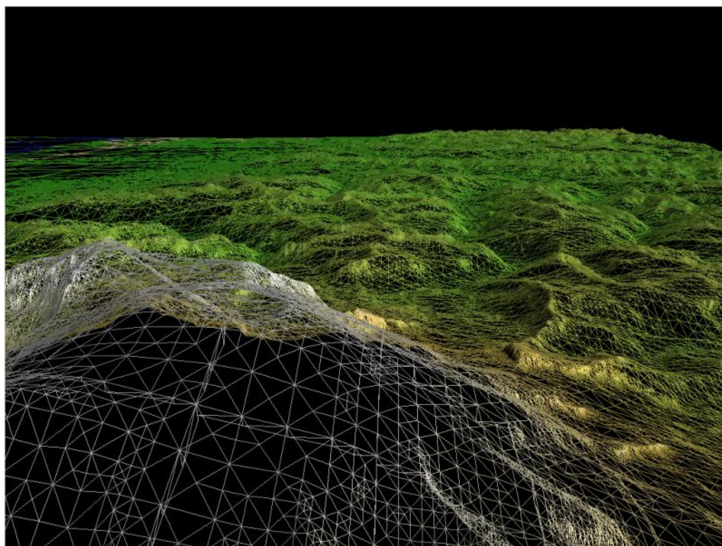
Jednym z największych problemów podczas renderowania cieni tą metodą w scenach typu *Outdoor* jest jednakowa rozdzielczość mapy cienia względem współrzędnych globalnych świata. Oznacza to, że chcąc rzucać na cały widoczny obszar sceny cień od światła typu kierunkowego (ang. *Directional Light*), np. od Słońca, „ząbek" odpowiadający pojedynczemu tekselowi mapy cienia ma na ekranie bardzo mały rozmiar dla miejsc odległych od kamery (na końcu pola widzenia), a bardzo duży dla miejsc w pobliżu kamery. Dlatego powstały metody reparametryzacji, które dodają do przekształcenia stosowanego przy mapowaniu cienia dodatkową, nieliniową transformację (np. perspektywiczną), dzięki której więcej miejsca na mapie cienia poświęcone zostaje obszarowi bliskiemu w stosunku do obserwatora, a mniej obszarom odległym. Podstawowym rozwiązaniem tego rodzaju jest ***Perspective Shadow Maps*** (PSM) [33]. Zobacz rys. 1.4. Ze względu na trudność w implementacji i pewne problemy powstały modyfikacje tej metody — *Light Space Perspective Shadow Maps* (LiSPSM) [34], *Trapezoidal Shadow Maps* (TSM) [35] oraz *Extended Perspective Shadow Maps* (XPSM) [36].



Rys. 1.4. *Perspective Shadow Mapping*: Po lewej pokazana mapa cienia, po prawej scena wyrenderowana z jej użyciem. (Źródło: [33])

**Renderowanie terenu** Renderowanie przestrzeni otwartych znajduje zastosowania w wielu aplikacjach wizualizacyjnych oraz w pewnych gatunkach gier (np. gry strategiczne, gry CRPG). Doskonałym punktem startowym do zdobywania wiedzy na temat przestrzeni otwartych w grafice komputerowej (we wszelkich aspektach — od terenu poprzez jaskinie, niebo, wodę, aż po drzewa, trawę i budynki) stanowi strona internetowa `vterrain.org` [37].

Podstawowym składnikiem sceny typu otwartego jest teren, czyli pofalowana powierzchnia odzwierciedlająca wzniesienia usypane z ziemi. Teren renderowany jest zwykle jako mapa wysokości (ang. ***Heightmap***, patrz rys. 1.5), tj. taka siatka trójkątów, w której wierzchołki są rozmieszczone równomiernie w płaszczyźnie poziomej



Rys. 1.5. Teren z widoczną siatką, z której jest zbudowany. (Źródło: [37])

i przesunięte w osi pionowej stosownie do wzorca ukształtowania terenu, pobieranego najczęściej z jasności pikseli specjalnej tekstury nazywanej właśnie mapą wysokości.

Wynalezione zostało wiele algorytmów wydajnego renderowania terenu. Niektóre z nich opisane są w książce traktującej wyłącznie na ten temat [38]. Jedną z najprostszych takich technik jest ***Geomipmapping*** [39]. Bardzo dobrym, ponieważ skutecznym a zarazem banalnie prostym rozwiązaniem problemu pęknięć (ang. *Cracks*) powstających w tej technice jest zastosowanie „spódniczki" (ang. *Skirt*) opisanej w [40]. Inny algorytm renderowania terenu to **ROAM** — ang. *Realtime Optimally–Adapting Meshes* [41]. Jeszcze innym, stosunkowo zaawansowanym rozwiązaniem w tym zakresie jest ***Clipmapping*** [42].

Niektóre z tych technik nienajlepiej współpracują ze współczesnym sprzętem graficznym, ponieważ operują na pojedynczych trójkątach. Wynalezione wiele lat temu, kiedy karty graficzne miały dużo mniejszą wydajność, teraz wykonują na CPU nadmiernie dużo obliczeń. Współczesne karty graficzne są zdolne do renderowania ogromnej liczby trójkątów na sekundę, jednakże do uzyskania jak najwyższej wydajności ważne jest podawanie tej geometrii dużymi porcjami (ang. *Batch*). W przeciwnym wypadku spada wydajność renderowania.

Do realistycznego wyrenderowania terenu potrzebne jest, oprócz ukształtowania siatki, także odpowiednie pokrycie teksturą. To również jest wyzwaniem, jako że rozległy teren nie powinien posiadać jednej tekstury, ale wiele tekstur płynnie między sobą przechodzących (np. śnieg, trawa, piasek, ziemia). Najpopularniejszym rozwiązaniem tego problemu jest ***Texture Splatting*** [43].

Renderowanie rozległego terenu rodzi dodatkowy problem wydajnego ograniczania zakresu renderowanej geometrii do tej pozostającej w polu widzenia. Stosowane bywają do tego techniki podziału przestrzeni takie jak drzewo czwórkowe (ang. *Quadtree*).

Innym problemem, jaki szczególnie daje o sobie znać podczas renderowania przestrzeni otwartych jest konieczność dynamicznej zmiany poziomu szczegółowości renderowanej geometrii wraz z odległością (**LOD** — ang. *Level of Detail*). O ile w przestrzeniach zamkniętych liczba widocznych obiektów jest zwykle znacznie ograniczona (niekiedy do zaledwie kilku) przez ściany wąskich korytarzy i ciasnych pomieszczeń, o tyle w otwartym terenie widoczne mogą być jednocześnie nawet setki albo tysiące obiektów takich jak drzewa czy jakieś postacie. Rozwiązaniem dla wydajnego renderowania takiej sceny jest właśnie LOD — wyświetlenie z mniejszą szczegółowością tych obiektów i tych części terenu, które znajdują się daleko od kamery. LOD można stosować do różnych aspektów renderowania grafiki, np. złożoności siatki trójkątów, dokładności obliczeń oświetlenia, płynności animacji, a nawet szczegółowości obliczeń fizyki czy sztucznej inteligencji.

Przełączanie się poziomów szczegółowości fragmentów terenu w miarę ich przybliżania lub oddalania od kamery powoduje nieprzyjemny efekt „przeskakiwania" (ang. *Popping*). Można go zniwelować poprzez zastosowanie **CLOD** — ang. *Continuous Level Of Details*.

**Renderowanie drzew i trawy** Wraz z postępem w grafice komputerowej, coraz częściej prezentowane są nie tylko budynki, maszyny i inne sztuczne twory, ale również dużo trudniejsze do realistycznego wyrenderowania obiekty naturalne takie jak rośliny. Sposób renderowania **drzew** przeszedł długą ewolucję. Pierwsze drzewa pokazywane były zwykle jako „plakaty" (ang. *Billboard*), czyli płaskie bitmapy zwrócone zawsze przodem do kamery. Innym częstym rozwiązaniem było złożenie trójwymiarowego obrazu drzewa z kilku przecinających się, oteksturowanych prostokątów. Wraz ze wzrostem mocy obliczeniowej kart graficznych pojawiła się możliwość bardziej dokładnego modelowania drzew wraz ze szczegółami takimi, jak kształt pnia, gałęzi i poszczególne grupy liści (choć nie były to — i wciąż nie są — pojedyncze liście).

Ręczne tworzenie modelu takiego drzewa przez artystę wymaga jednak dużo pracy. Inną wadą tego podejścia jest pracochłonność otrzymania wielu podobnych, ale niejednakowych modeli. Dlatego wirtualne drzewa są jednym z tych obszarów grafiki, w których proceduralne generowanie sprawdza się równie dobrze, a nawet lepiej niż jego kreowanie przez artystę. Powstało wiele rozwiązań skupiających się czy to na algorytmach generowania kształtu drzew (pnia, gałęzi, liści), czy też na ich wydajnym renderowaniu w czasie rzeczywistym. Spośród tych pierwszych na uwagę zasługuje przede wszystkim [44]. Podczas opracowania metody renderowania drzew nieocenioną inspiracją może być studiowanie wyglądu i działania technologii SpeedTree [45] (szczególnie dostępnego do pobrania za darmo dema Trees of Pangaea), która skupia się ściśle na generowaniu i renderowaniu realistycznych, proceduralnych drzew oraz jest licencjonowana na potrzeby wielu współczesnych gier i innych aplikacji. Zobacz rys. 1.6.




Rys. 1.6. Przykład generowania i renderowania fotorealistycznej roślinności w czasie rzeczywistym. (Źródło: [45])

Osobnym, niebanalnym zagadnieniem jest wydajne renderowanie **trawy**. Źdźbła trawy występują w ogromnej ilości pokrywając siatkę terenu, co wymaga specjalnego potraktowania tematu optymalizacji ich renderowania. Pewnie rozwiązania prezentują artykuły [46, 47]. Inne, bardzo zaawansowane podejście oparte na trzypoziomowym LOD opracowane zostało przez badaczy z INRIA [48].

**Inne efekty przestrzeni otwartych** Podczas renderowania przestrzeni otwartych zachodzi konieczność prezentowania w jakiś sposób także sklepienia niebieskiego, na które składają się elementy takie jak: chmury, Słońce, gwiazdy i inne ciała niebieskie. Najprostsza metoda to tzw. *Skybox* — ogromny sześcian poruszający się wraz z kamerą (ale nie obracający się wraz z nią), oteksturowany od środka za pomocą tekstury sześciennej przedstawiającej niebo wraz z chmurami, Słońce, teren na horyzoncie itd. Realistyczne renderowanie dynamicznie zmieniającego się nieba **nieba** jest tematem wielu publikacji. Większość z nich skupia się na konkretnym zagadnieniu, takim jak realistyczny dobór kolorów tła czy model ruchu ciał niebieskich.

Wdzięcznym tematem badań są techniki renderowania **chmur**. Najprostsza metoda polega na prezentowaniu przesuwających się, płaskich fotografii przedstawiających chmury. Do proceduralnego generowania realistycznych formacji tego typu doskonale nadaje się szum Perlina [49]. Możliwe są także inne, bardziej zaawansowane podejścia. Na przykład w zastosowaniach, w których konieczne jest wlatywanie wirtualną kamerą do wnętrza chmur, a nie tylko oglądanie ich z powierzchni Ziemii (jak w symulatorze lotu Microsoft Flight Simulator) zastosowanie mogą znaleźć metody takie jak opisana w [50]. Zobacz rys. 1.7.

Realizmu przestrzeniom otwartym dodają efekty atmosferyczne — opady deszczu, śniegu czy burza piaskowa (zależnie od rodzaju wirtualnej krainy). W tej materii, po-



Rys. 1.7. Rendering fotorealistycznych chmur w czasie rzeczywistym. (Źródło: [50])

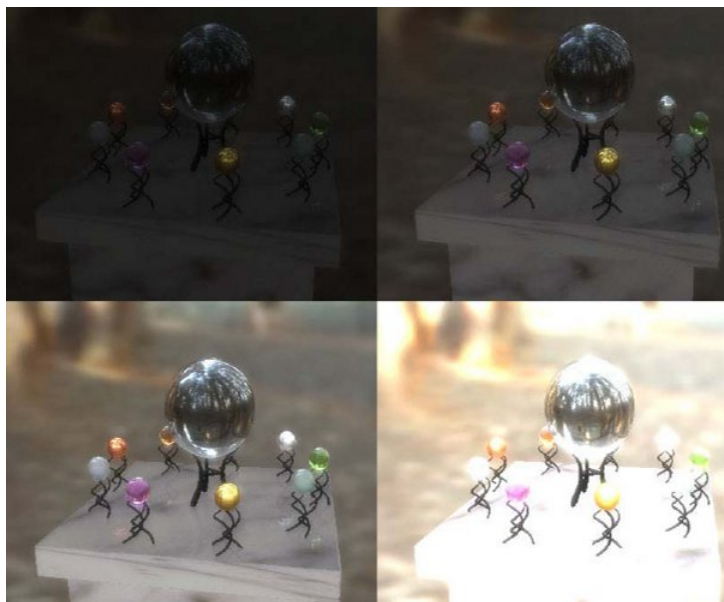
dobnie jak w całej grafice komputerowej, każdy doświadczony programista jest w stanie wymyślić własne oryginalne rozwiązania. Niezwykle zaawansowane i kompleksowe podejście do wszelkich efektów związanych z opadami deszczu prezentuje [51].

Podobnie jest z powierzchnią wody. Woda jest trudna do realistycznego przedstawienia, ponieważ wymaga zamodelowania wielu zjawisk optycznych takich jak refleksja i refrakcja światła, kaustyki (ang. *Caustics*) itd. Wymaga ponadto pokazywania zarówno tego, co znajduje się pod powierzchnią jak i obrazu odbitego od powierzchni. Dlatego jakość renderowanej wody często bywa poświęcana na rzecz prostoty obliczeń (a co za tym idzie — wydajności). Nic więc dziwnego, że w praktycznie każdej grze spotkać można inaczej wyglądającą, inaczej zrealizowaną wodę. Bardzo dobra, zaawansowana implementacja renderowania wody znajduje się w [52].

**Efekty postprocessingu** *Postprocessing* (co można przetłumaczyć jako „po–przetwarzanie") to grupa efektów nakładanych na gotowy już obraz w sposób dwuwymiarowy. Istnieje bardzo wiele takich efektów. Autor wybrał i zaimplementował w swoim kodzie niektóre z nich.

**HDR** (ang. *High Dynamic Range*) to pojęcie wprowadzone przez [53]. Oznacza szeroki zakres jasności, jaka występuje w naturze i jaka jest obserwowalna przez ludzkie oko. Zarówno współczesne kamery i aparaty fotograficzne, jak i monitory czy drukarki nie są w stanie oddać tak szerokiego zakresu jasności. Dlatego realizm wirtualnych scen można ulepszyć uwzględniając zjawisko HDR w procesie renderowania. Wykonanie pełnego renderingu HDR wymaga specjalnego podejścia do całego procesu, m.in. wykorzystania tekstur w formacie zmiennoprzecinkowym lub RGBE (*Red, Green, Blue, Exponent*) celem odwzorowania szerszego zakresu jasności, niż oferuje to standardowy format piksela typu `A8R8G8B8` (który przeznacza tylko 8 bitów na skła-

dową). Można jednakże spróbować zastosować efekty nakładane na koniec procesu renderowania HDR bez wykorzystywania prawdziwego HDR. Powstaje wówczas tzw. *Fake HDR*, który wciąż polepsza jakość i realizm renderowanej sceny, mimo swojej prostoty. Te efekty to *Tone Mapping* [54] i *Bloom* [55]. Zobacz rys. 1.8.



Rys. 1.8. *Tone Mapping* z różnymi wartościami ekspozycji. (Źródło: [55])

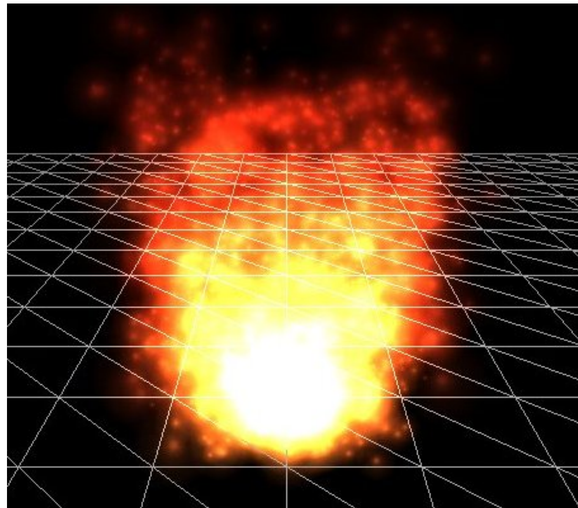
Sprzężenie zwrotne (ang. ***Feedback***) [3] to nakładanie poprzedniej klatki obrazu na nową w sposób półprzezroczysty. Zastosowanie znajduje np. w wizualizacjach muzycznych, takich jak wtyczka AVS w odtwarzaczu Winamp. W grafice 3D może posłużyć do otrzymania różnorodnych efektów, jak np. proste rozmycie ruchu (ang. *Motion Blur*) dla całego obrazu.

Błyski soczewek (ang. ***Lens Flare***) to prosty efekt powstający w wyniku zjawisk optycznych, jakie zachodzą na soczewkach obiektywu kamery, kiedy wpada do niej światło prosto od Słońca lub innego silnego źródła. W naturze efekt ten jest niepożądany, ale w grafice komputerowej może dodać scenie realizmu podkreślając jaskrawość źródła światła, niemożliwą do oddania na ekranie monitora bezpośrednio. Sposób implementacji tego efektu opisuje np. [3].

Mgła ciepła (ang. ***Heat Haze***) to z kolei zjawisko falowania obrazu załamującego się w powietrzu ogrzewanym np. przez rozgrzany asfalt jezdni albo płomień. Sposób implementacji tego efektu opisuje [56].

**Efekty specjalne** Pewne zjawiska są wolumetryczne, czyli posiadające charakter przestrzenny. Najpopularniejszym efektem tego rodzaju jest **efekt cząsteczkowy** (ang. *Particle Effect*). Zobacz rys. 1.9. Jego realizacja polega na renderowaniu zbioru punktów (cząsteczek), z których każdy staje się w procesie renderowania kwadratem zwróconym zawsze prostopadle do kierunku patrzenia kamery, pokrytym półprzezroczystą teksturą. Dzięki wykorzystaniu odpowiedniej tekstury, odpowiednich ustawień

blendingu (często używany bywa tutaj blending addytywny) oraz odpowiednio dużej liczby cząsteczek, małym nakładem pracy uzyskać można niezwykle efektowne animacje przedstawiające np. ogień, dym, iskry czy jakieś fantastyczne, magiczne drobinki „energii”.



Rys. 1.9. Efekt cząsteczkowy zastosowany do renderowania ognia. (Źródło: Wikipedia)

Efekty cząsteczkowe podzielić można na stanowe i bezstanowe [57]. **Efekty cząsteczkowe bezstanowe** polegają na wyliczaniu wszystkich parametrów cząstki (jak pozycja, rozmiar, kolor, orientacja) w funkcji czasu ze ściśle określonego wzoru. To pozwala na ich łatwą i wydajną implementację realizowaną w pełni na GPU. Narzuca jednocześnie ograniczenie na ruch tych cząstek do pewnego stałego scenariusza. **Efekty cząsteczkowe stanowe** polegają na aktualizowaniu parametrów cząstek zgodnie z krokiem czasowym na podstawie ich stanu poprzedniego. Daje do dużo większą elastyczność w manipulowaniu tymi parametrami. Możliwe jest np. obliczanie kolizji i odbić cząstek od geometrii mapy. Pojawiły się próby implementacji efektów cząsteczkowych stanowych przeliczanych na GPU, co stało się możliwe dzięki rosnącym możliwościom i elastyczności nowoczesnego sprzętu graficznego.

Istnieje dużo innych efektów specjalnych, dla których wynalezione zostały użyteczne modele i wydaje implementacje z użyciem akceleracji sprzętowej, a które nie zostały opisane w niniejszej pracy. Należą do nich np. mgła wolumetryczna (ang. *Volumetric Fog*) czy snopy światła (ang. *Light Shaft*).

## 1.2. Charakterystyka dostępnych bibliotek graficznych

Z układem scalonym karty graficznej bezpośrednio komunikuje się przeznaczony dla niej sterownik. Jednakże dla programisty udostępniony zostaje ujednolicony interfejs, niezależny od tego, kto jest producentem układu graficznego (np. NVIDIA, ATI/AMD, Intel) ani też jak nowoczesna jest to karta. Ów interfejs na platformie PC ma postać jednej z dwóch bibliotek: DirectX lub OpenGL.

Twórcą biblioteki DirectX jest firma Microsoft [3]. Jest ona przeznaczona na platformę PC z systemem Windows oraz konsolę XBox (której nazwa pochodzi właśnie od ang. *DirectX–Box*). Wersje DirectX kompatybilne z Windows XP i Vista noszą obecnie oznaczenie 9.0c i są dodatkowo oznaczane datą wydania konkretnej wersji (np. *March 2008*), a nowe aktualizacje ukazują się kilka razy w roku. Programowanie z użyciem DirectX wymaga zainstalowanego pakietu DirectX SDK w wybranej wersji. Od użytkownika natomiast wymagane jest, aby w swoim systemie posiadał zainstalowaną bibliotekę DirectX (tzw. *DirectX Redistributable*) odpowiadającą tej wersji SDK, w której program został skompilowany (lub nowszą).

Interfejs biblioteki DirectX jest obiektowy, oparty na technologii COM. Począwszy od wersji 8 nastąpiła jego znaczna reorganizacja i uproszczenie. Zniknął ścisły podział na część przeznaczoną do grafiki 2D i 3D (określane jako DirectDraw i Direct3D). DirectX jest kompletną biblioteką multimedialną, która oprócz części graficznej (Direct3D) wspiera też m.in. odtwarzanie dźwięku oraz obsługę urządzeń wejściowych (klawiatura, mysz i różnego rodzaju manipulatory, np. pady, kierownice). W skład DirectX wchodzi rozszerzenie D3DX, które stanowi bogatą bibliotekę funkcji matematycznych (zapewniającą operacje na wektorach, macierzach, kwaternionach, płaszczyznach itp.), a także potrafiących wczytywać tekstury z plików w różnych formatach graficznych (m.in. BMP, JPEG, PNG, TGA, DDS), siatki modeli w formacie X i wielu innych. Językiem shaderów używanym w DirectX jest HLSL (ang. *High Level Shading Language*).

Najnowszą wersją biblioteki jest DirectX 10. Jest ona niekompatybilna wstecz. Jej intefejs został całkowicie zreorganizowany, uproszczony, a wiele przestarzałych elementów zostało porzucone. Nowa wersja daje dostęp do nowych funkcji kart graficznych najnowszej generacji (w chwili pisania tych słów), tj. GeForce 8000/9000 (i ich odpowiedników firmy ATI/AMD), m.in. Shader Model 4. Niestety, DirectX 10 do działania wymaga posiadania takiej karty, jak również systemu Windows Vista, gdyż nie ma wersji DirectX 10 dla Windows XP. To czyni najnowszą wersję biblioteki nienajlepszym wyborem. W chwili, kiedy takie karty graficzne są jeszcze stosunkowo drogie i niezbyt rozpowszecnione, a przejście na Windows Vista napotyka dużą niechęć użytkowników, pisanie aplikacji używających wyłącznie DirectX 10 byłoby znacznym ograniczeniem grona potencjalnych odbiorców.

Drugą z bibliotek dających dostęp do możliwości karty graficznej jest OpenGL (ang. *Open Graphics Library*) [4]. API to, stworzone przez Silicon Graphics Inc. (SGI), jest obecnie rozwijane przez grupę Khronos. Jest przenośne — pozwala na pisanie aplikacji działających zarówno w Windows, jak i w Linux oraz na innych platformach. Nie wymaga od użytkownika instalacji żadnego oprogramowania — jest dostępny standardowo w systemie Windows.

OpenGL w powszechnej opinii uchodzi za bibliotekę prostszą do opanowania i przez to lepiej nadającą się do nauki programowania grafiki 3D dla początkujących. Owo

wrażenie jest jednak złudne, gdyż programista po pewnym czasie siłą rzeczy napotyka na przeszkody takie jak konieczność samodzielnego manipulowania wektorami i macierzam (która w Direct3D występuje od początku), a OpenGL w żaden sposób tego nie ułatwia (w przeciwieństwie do Direct3D, wyposażonego w dodatek D3DX). Interfejs OpenGL jest strukturalny, oparty na stanach. Z możliwości nowoczesnych kart graficznych korzysta się w nim przede wszystkim za pomocą tzw. rozszerzeń. W planach jest nowa wersja OpenGL 3.0, której API ma zostać zupełnie przeorganizowane i dostosowane do funkcjonalności najnowszych układów graficznych, podobnie jak Microsoft zrobił to w nowym DirectX 10.

Biblioteki Direct3D i OpenGL mają niemal takie same możliwości oraz praktycznie taką samą wydajność. Dlatego nie sposób rozstrzygnąć jednoznacznie, która z nich jest lepsza. W obydwu można zaimplementować te same efekty graficzne. Obydwie te biblioteki są darmowe do wykorzystania zarówno dla programisty, jak i dla użytkownika. Jednakże faktem jest, że w chwili obecnej większość gier na PC powstaje przeznaczona dla systemu Windows i używa biblioteki DirectX.

Biorąc pod uwagę powyższe kryteria, jak również osobiste preferencje autora, jako biblioteka graficzna wykorzystana podczas implementacji silnika, który jest przedmiotem tej pracy, wybrany został DirectX 9.0c.

## 1.3. Cel i zakres pracy

Celem pracy jest stworzenie biblioteki programowej zwanej silnikiem grafiki trójwymiarowej, z użyciem języka programowania C++ i biblioteki graficznej DirectX. W ten sposób autor chciał poznać, jak wygląda architektura i implementacja takiego silnika.

Zakres pracy obejmuje:

- badania literaturowe w dziedzinie renderowania grafiki trójwymiarowej,
- badanie algorytmów i technik renderowania różnorodnych efektów graficznych,
- badanie architektury silników graficznych 3D i rozważania nad problemami, które się przy tym pojawiają,
- omówienie modułów i klas, z jakich składa się silnik graficzny i kod towarzyszący, na którym się on opiera,
- opracowanie przykładów zastosowania silnika (prototypów gier),
- przeprowadzenie testów zaimplementowanego systemu,
- analiza wydajności systemu na dostępnym współcześnie sprzęcie graficznym.

Stworzony na potrzeby tej pracy kod nie może oczywiście dorównać możliwościami ani jakością silnikom rozwijanym przez wiele lat i przez wielu ludzi, czy to tworzonym w środowisku Open Source (jak Ogre3D, Irrlicht) czy też przez duże firmy z branży gier komputerowych (jak Unreal Engine firmy Epic Games czy id Tech 5 firmy id Software). Stanowi jednak względnie efektowny, kompletny oraz — co bardzo ważne

— ukończony projekt mogący teoretycznie znaleźć zastosowanie przy tworzeniu gry komputerowej lub symulacji w czasie rzeczywistym.

Niniejsza praca ma charakter przeglądowy. W skład silnika graficznego wchodzi bardzo wiele różnych technik, algorytmów i innych rozwiązań, dlatego nie sposób opisać ich wszystkich dokładnie. Każda z nich mogłaby być tematem osobnej rozprawy. Ponadto celem niniejszej pracy nie jest wprowadzenie jakiegokolwiek zupełnie nowego rozwiązania, a raczej pokazanie, w jaki sposób można połączyć wybrane istniejące rozwiązania aby zrealizować założony cel. Aczkolwiek przy implementacji tak rozbudowanego projektu programistycznego, siłą rzeczy zastosowane zostało wiele drobnych, nowatorskich pomysłów.

W rozdziale 1 niniejszej pracy określona zostaje ściśle dziedzina, której praca dotyczy oraz wprowadzone zostaje pojęcie silnika graficznego. Przedstawiony jest również przegląd znanych technik realizacji poszczególnych efektów graficznych. W rozdziale 2 wprowadzony zostaje podział przedstawionego systemu na warstwy i opisane są poszczególne moduły tych warstw, składające się na nie klasy i inne elementy kodu. W ten sposób pokazana zostaje architektura silnika. W rozdziale 3 opisane są dokładniej wybrane elementy silnika. Za pomocą rysunków, listingów w pseudokodzie i fragmentów shaderów w języku HLSL przedstawiony zostaje sposób realizacji wybranych efektów graficznych zawartych w silniku, ich algorytmy i obliczenia, jakie im towarzyszą. Rozdział 4 zawiera kolorowe zrzuty ekranu prezentujące efekt końcowy działania poszczególnych elementów silnika. W rozdziale 5 znajduje się podsumowanie oraz wnioski końcowe, a także informacje o możliwościach rozbudowy przedstawionego silnika w przyszłości.

# Rozdział 2

# Architektura silnika

System opisany w tej pracy podzielony jest na wyraźnie rozróżnione warstwy zgodnie z pewnymi zasadami. Rysunek 2.1 przedstawia strukturę kodu systemu. Każda warstwa niższa jest bardziej podstawowa i nie korzysta z warstw wyższych, które są bardziej wyspecjalizowane. Każda warstwa może korzystać tylko z warstw niższych. Dzięki temu warstwy bardziej podstawowe mogą być wykorzystywane bez warstw wyższego poziomu, co autor niejednokrotnie wykorzystał pisząc aplikację konsolową wyłącznie z użyciem modułów bazowych czy też grę dwuwymiarową wyłącznie z użyciem modułów bazowych i szkieletu, bez silnika. Fizycznie każda z warstw posiada kod i dane zgromadzone w osobnym katalogu.




Rys. 2.1. Struktura kodu systemu opisanego w tej pracy.

Charakterystyka poszczególnych warstw systemu:

- **Biblioteka bazowa** nosi w kodzie nazwę `Common`. Jest to zbiór plików źródłowych w C++ zapewniających podstawowe funkcje i klasy przydatne podczas programowania różnego rodzaju aplikacji, w tym obsługę łańcuchów i konwersji, moduł matematyczny, hierarchię klas wyjątków do obsługi błędów, moduł do daty i czasu, moduł do programowania wielowątkowego, hierarchię klas strumieni, obsługę systemu plików czy logger. w przeciwieństwie do pozostałej części systemu, biblioteka modułów bazowych jest przenośna między systemami Windows

i Linux.

- **Szkielet** nosi w kodzie nazwę `Framework`. Jest to kod zapewniający podstawową funkcjonalność aplikacji wykorzystującej renderowanie grafiki czasu rzeczywistego w DirectX. Obejmuje zagadnienia takie jak tworzenie okna Windows i inicjalizacja biblioteki Direct3D, obsługę wejścia z klawiatury i myszki, a także konsolę tekstową, manager zasobów wraz z hieararchią klas różnego rodzaju zasobów Direct3D, moduł do renderowania grafiki 2D oraz system GUI.
- **Silnik** nosi w kodzie nazwę `Engine`. Jest to najważniejsza i najbardziej złożona część całego systemu. Stanowi spójny podsystem zapewniający renderowanie grafiki 3D i tworzący w tym celu warstwę abstrakcji ponad bibliotekę Direct3D, dzięki której użytkownik może się posługiwać klasami reprezentującymi abstrakcyjne pojęcia, takie jak scena, kamera, materiał czy encja, bez zajmowania się szczegółami implementacyjnymi.
- **Gra** nosi w kodzie nazwę `Client`. Jest to najwyższa warstwa, implementująca logikę aplikacji i korzystająca w tym celu z funkcjonalności zapewnianej przez silnik. w kodzie dołączonym do pracy stanowią ją proste prototypy 5 gier różnego gatunku, których jedynym celem jest pokazanie możliwości graficznych silnika. Własna implementacja tej warstwy jest podstawowym zadaniem programisty, który ma zamiar użytkować silnik.
- **Narzędzia** noszą w kodzie nazwę `Tools`. Jest to zbiór dodatkowych procedur przetwarzających różnego rodzaju dane potrzebne do pracy silnika. W kodzie dołączonym do pracy narzędzia te mają postać pojedynczej aplikacji konsolowej uruchamianej z wiersza poleceń z odpowiednimi parametrami (nie posiadającej interfejsu graficznego). Aplikacja ta potrafi przetwarzać tekstury, modele oraz mapy.
- **Eksportery** noszą w kodzie nazwę `Plugins`. Jest to zbiór dodatkowego oprogramowania dołączanego do zewnętrznych narzędzi w celu zapewnienia kompatybilności z formatami plików używanymi przez silnik. W dołączonym kodzie eksportery te mają postać wtyczek napisanych w języku Python, przeznaczonych do eksportowania grafiki trójwymiarowej z darmowego programu Blender to formatów pośrednich, przetwarzanych potem na formaty docelowe przez wspomniane wyżej narzędzia konsolowe.

## 2.1. Podstawowe założenia

Jako platforma wybrany został komputer typu **PC** i system **Windows**. Decyzja ta podyktowana była głównie osobistymi preferencjami autora, aczkolwiek jest też uzasadniona w szerszym kontekście. O ile system Linux znajduje szerokie zastosowania w wielu dziedzinach informatyki, o tyle w branży gier komputerowych jego znaczenie jest niewielkie. Coraz większą rolę odgrywają natomiast w tej branży inne platformy

sprzętowe — konsole, np. PlayStation 3, XBox360, Wii czy urządzenia przenośne. Jednakże pakiety programistyczne do nich nie są często dostępne publicznie. Dlatego platforma Windows jest dobrym wyborem. Należy przy tym dodać, że nowoczesne, rozbudowane silniki są często pisanie w sposób pozwalający na ich działanie na wielu platformach — w szczególności na PC oraz na konsolach. Silnik napisany na potrzeby tej pracy nie jest przenośny na inne platformy, co uprościło jego implementację.

Bardzo ważne było określenie możliwości sprzętu graficznego, jakie zostaną użyte w silniku. Wybrany został **Shader Model 2**, dostępny na kartach począwszy od GeForce 5 (FX) w górę i ich odpowiednikach firmy ATI. Sprzęt graficzny rozwija się bardzo szybko i stosunkowo niewiele czasu potrzeba, aby nowe generacje chipów graficznych stały się standardem pośród domowych użytkowników. Równocześnie nadal wiele jest komputerów posiadających stare lub bardzo stare karty graficznej. Są to przede wszystkim komputery biurowe i domowe należące do tych użytkowników, którzy nie są pasjonatami gier. Rozsądne wymagania w tym względzie, jakie może mieć gra, zależą więc od rodzaju tej gry. Po produkcie najwyższej jakości z bardzo realistyczną grafiką można spodziewać się wysokich wymagań sprzętowych, podczas gry małe typu *Casual* powinny mieć raczej niskie wymagania sprzętowe. Trzeba dodać przy tym, że dobre silniki potrafią się zwykle dostosować do sprzętu, na którym zostają uruchomione wyłączając niektóre efekty bądź też wybierając osobną, specjalnie przygotowaną dla niższej klasy sprzętu ścieżkę renderowania. Silnik napisany na potrzeby tej pracy nie posiada takiej możliwości (korzysta wyłącznie z Shader Model 2 i jego wymaga), co uprościło jego implementację.

Jako język programowania wybrany został **C++**. O ile w informatyce ogólnie używane bywają różne języki programowania i wiele z nich jest dużo popularniejszych i częściej stosowanych w pewnych dziedzinach (jak Java i C# w aplikacjach biznesowych), o tyle w branży gier komputerowych decyzja o wyborze języka C++ jest oczywista i nie wzbudza żadnych kontrowersji. Język ten pozwala programować w sposób względnie wygodny i na wysokim poziomie (dzięki wsparciu dla programowania obiektowego i stosunkowo rozbudowanej bibliotece standardowej), a równocześnie jest kompilowany do kodu natywnego pozwalając uzyskać najwyższą wydajność i najlepsze wykorzystanie sprzętu, co ma pierwszorzędne znaczenie.

Jako środowisko programistyczne wybrany został **Microsoft Visual Studio 2005 Professional**. Środowisko to jest bardzo często używane przez profesjonalistów programujących w C++ i uchodzi za najlepsze jakie istnieje na platformie Windows dzięki wygodnemu edytorowi, debuggerowi i dobremu kompilatorowi.

Jako biblioteka graficzna wybrany został **DirectX 9.0c**. Na platformie Windows istnieją dwie biblioteki graficzne pozwalające na renderowanie grafiki trójwymiarowej z użyciem akceleracji sprzętowej — DirectX i OpenGL. Obydwie mają taką samą wydajność i takie same możliwości (w praktyce udostępniają po prostu funkcjonalność karty graficznej). Obydwie są darmowe do użycia zarówno dla programisty, jak i użyt-

kownika. Tak więc wybór jednej z nich jest raczej kwestią osobistych preferencji. Aczkolwiek podkreślić należy, że w zastosowaniu do poważnych gier na platformie PC częściej stosowany jest DirectX.

Ponadto do napisania kodu przedstawionego silnika użyty został szereg innych narzędzi, w tym: GIMP (edytor grafiki bitmapowej), Blender (edytor grafiki 3D), Bitmap Font Generator (narzędzie do generowania tekstur przedstawiających znaki wybranej czcionki tekstowej), jEdit (edytor tekstu dla programistów), SVN (system kontroli wersji).

## 2.2. Biblioteki zewnętrzne

Do implementacji opisywanego systemu użyte zostały następujące biblioteki zewnętrzne:

- **Biblioteka standardowa C++**, w szczególności łańcuchy znaków `std::string` i kontenery STL takie jak `std::vector`, `std::list`, `std::map`,
- **API systemowe**, czyli funkcje udostępniane przez system operacyjny (przede wszystkim z nagłówka `windows.h`),
- **FastDelegate** — biblioteka realizująca brakujący w języku C++, a bardzo potrzebny w niektórych zastosowaniach element składniowy — delegaty (ang. *Delegate*, nazywane też wskaźnikami na składowe — ang. *Pointer to Member*, sygnałami i slotami — ang. *Signal, Slot* albo wywołaniami zwrotnymi — ang. *Callback*), czyli wskaźniki na metody obiektów,
- **NVMeshMender** — biblioteka firmy NVIDIA służąca do obliczania wektorów normalnych i stycznych w siatkach trójkątów,
- **zlib** — biblioteka do kompresji danych w pamięci algorytmem Deflate oraz obsługi formatu pliku GZip.

## 2.3. Architektura biblioteki bazowej

Biblioteka bazowa to najniższa warstwa systemu. Stanowi kod w C++ zgromadzony w katalogu `Common`, w parach odpowiadających sobie plików źródłowych `CPP` i nagłówkowych `HPP`. Każda taka para stanowi osobny moduł, przy czym niektóre moduły są ze sobą powiązane — wymagają do działania innych. Listę modułów podstawia tabela 2.1.

Biblioteka ta jest niezależna od całej reszty opisywanego systemu. Stanowi kod przydatny podczas pisania w języku C++ różnego rodzaju programów, choć powstała głównie z myślą o programowaniu gier. Jej aktualna wersja, oznaczona numerem iteracji 7, powstawała od połowy 2006 roku.

Biblioteka bazowa została też opublikowana w Internecie za darmo, na licencji GNU LGPL, pod nazwą **CommonLib**.

Tablica 2.1. Lista modułów biblioteki bazowej.

| | |
|---|---|
| Base | Moduł podstawowy |
| Config | Obsługa plików konfiguracyjnych |
| DateTime | Obsługa daty i czasu |
| Dator | Tekstowy dostęp do zmiennych różnego typu |
| Error | Hierarchia klas wyjątków do obsługi błędów |
| Files | Obsługa systemu plików |
| FreeList | Szybki alokator pamięci |
| Logger | Rozbudowany mechanizm logowania komunikatów |
| Math | Biblioteka matematyczna |
| Profiler | Klasa służaca do mierzenia wydajności |
| Stream | Hierarchia klas strumieni binarnych |
| Threads | Biblioteka do wielowątkowości i synchronizacji |
| Tokenizer | Prosty parser plików tekstowych |
| ZlibUtils | Obiektowa otoczka na bibliotekę kompresji zlib |

**Podstawowe założenia** Większość z wymienionych założeń jest wspólna dla kodu całego systemu. W przeciwieństwie do reszty systemu, biblioteka jest przenośna między platformami Windows i Linux. Założenie to zostało podyktowane chęcią wykorzystania biblioteki także do napisania aplikacji serwerowej (serwera gry), która powinna działać na platformie uniksowej. Biblioteka ma niewiele zewnętrznych zależności. Używa wyłącznie biblioteki standardowej C++, API systemowego, a opcjonalny moduł ZlibUtils dodatkowo biblioteki zlib. Do przechowywania wszelkich łańcuchów tekstowych używany jest typ `std::string` z biblioteki standardowej C++. Nie jest to wybór optymalny ze względów wydajnościowych, ale zapewnia większą wygodę i bezpieczeństwo, niż używanie surowego wskaźnika do tablicy znaków typu `char*`. Do obsługi błędów używany jest mechanizm wyjątków C++ własnego typu, klas zapewnianych przez moduł Error. Biblioteka pisana jest z myślą o jak największej wydajności, ale nie kosztem bezpieczeństwa ani wygody używania. Korzysta z zaawansowanych możliwości języka C++ takich jak częściowa specjalizacja szablonów, dlatego wymaga dobrego kompilatora C++, zgodnego ze standardem. Testowana była na kompilatorach Visual C++ 2005 oraz GCC 4. Wszystkie elementy zgromadzone są w przestrzeni nazw `common`. Wersja biblioteki dołączona do pracy nie wspiera Unikodu. Polega na rozmiarze, a nawet budowie bitowej typów atomowych takich jak znaki, liczby całkowite i zmiennoprzecinkowe. Przez to nie nadaje się do użycia na platformach z inną kolejnością bajtów czy 64–bitowych.

**Moduł Base** Moduł bazowy to kod zgromadzony w plikach `Common\Base.hpp` i `Common\Base.cpp`. Nie używa on żadnego innego modułu, za to każdy inny moduł i każdy plik źródłowy całego systemu może i powinien używać jego włączając do kodu stosowny nagłówek. Zapewnia różnorodne możliwości, których zdaniem autora brakuje w bibliotece standardowej C++, a które są przydatne podczas programowania.

Poniższy opis nie wyczerpuje wszyskich elementów tego modułu, a jedynie te ważniejsze i bardziej godne uwagi. Pełną listę definiowanych symboli można znaleźć w pliku nagłówkowym.

Moduł włącza nagłówek `<string>` z biblioteki standardowej C++ i deklaruje `using std::string;` po to, aby typu łańcuchowego `string` można było używać wszędzie niczym typów wbudowanych.

Moduł definiuje także za pomocą `typedef` podstawowe typy całkowitoliczbowe o jawnie określonej długości: `int1`, `int2`, `int4`, `int8`, `uint1`, `uint2`, `uint4`, `uint8`. Przy intensywnym wykorzystaniu plików binarnych, a w przyszłości może także transmisji sieciowej nie sposób uciec od wyspecyfikowania długości poszczególnych danych liczbowych, jak to robi standard języka C++ definiując np. typ `int` jako odpowiadający długości słowa maszynowego na danej platformie. Wiedzą o tym twórcy nowoczesnych języków takich jak C# czy Java, gdzie długość poszczególnych typów liczbowych jest stała i określona.

Operator `absolute_cast` — ten prosty, sprytnie skonstruowany szablon funkcji imituje zachowanie operatorów rzutowania C++. Jego zadaniem jest uzupełnić brak w tym języku operatora, który pozwoliłby na dosłowną reinterpretację bitową wartości dowolnego typu na dowolny inny typ. Zarówno stare rzutowanie w stylu C, jak i nowe operatory `static_cast` i `reinterpret_cast` nie potrafią na przykład potraktować liczby typu `float` jako liczby typu `unsigned int` w sposób dosłowny, bez dokonywania konwersji numerycznej. Często stosowanym obejściem jest pobieranie adresu zmiennej, rzutowanie go na wskaźnik do innego typu i wyłuskanie otrzymanego wskaźnika. Równoważnym rozwiązaniem jest rzutowanie na referancję do innego typu.

Oczywiście, potrzeba użycia takiej konstrukcji zachodzi niezwykle rzadko i nie można uznać jej za szczyt elegancji w programowaniu, ale bywa przydatna jako sztuczka optymalizacyjna pozwalająca na przykład na szybkie generowanie pseudolosowych liczb zmiennoprzecinkowych. Jej stosowanie jest też konieczne podczas używania biblioteki Direct3D do podawania liczb zmiennoprzecinkowych do funkcji `IDirect3DDevice9::SetRenderState`.

```
template <typename destT, typename srcT>
destT & absolute_cast(srcT &v)
{
  return reinterpret_cast<destT&>(v);
}
```

Moduł dostarcza własnej implementacji inteligentnych wskaźników. Inteligentny wskaźnik (ang. *Smart Pointer*) to klasa, której zachowanie imituje wskaźnik, który jednak sam w swoim destruktorze zwalnia obiekt, na który wskazuje. Jest to przykład zastosowania wzorca RAII (ang. *Resource Acquisition Is Initialization*).

Implementacja inteligentnych wskaźników wzorowana jest na bibliotece Boost. Autor chciał uniknąć użycia tej biblioteki w swoim projekcie, dlatego wiele przedstawio-

nych tu elementów dubluje jej funkcjonalność. Zaprojektowanie implementacji tego mechanizmu wymagało podjęcia wielu trudnych decyzji projektowych. Ostatecznie ich budowa jest prosta, ale rozszerzalna o politykę (ang. *Policy* — rozwiązanie spopularyzowane przez Andrei Alexandrescu [58]) określającą sposób zwalniania obiektu.

Dostępne są cztery rodzaje inteligentnych wskaźników: `scoped_ptr`, `shared_ptr`, `scoped_handle`, `shared_handle`. Klasy typu „scoped–" posiadają wskazywany obiekt niejako na własność — nie pozwalają na kopiowanie wskaźnika i zawsze zwalniają go w destruktorze. Klasy typu „shared–" pozwalają na kopiowanie wskaźnika i posiadają licznik referencji. Wskazywany obiekt jest zwalniany, kiedy licznik referencji spada do zera (żaden inteligentny wskaźnik na niego nie wskazuje). Klasy typu „–ptr" przechowują wskaźnik do typu, którym są sparametryzowane. Klasy typu „–handle" przechowują bezpośrednio wartość typu, którym są sparametryzowane. Służą do przechowywania uchwytów, np. na otwarte pliki.

Dostępne są trzy predefiniowane polityki zwalniania wskaźników: `DeletePolicy` używa operatora `delete`. `DeleteArrayPolicy` używa operatora `delete[]` (przeznaczony jest do tablic). `ReleasePolicy` dokonuje wywołania `x->Release();`. Przeznaczone jest np. do obiektów COM. Ponadto dostępne są dwie predefiniowane polityki zwalniania uchwytów. `CloseHandlePolicy` wywołuje funkcję `CloseHandle(x);`, natomiast `DeleteObjectPolicy` wywołuje funkcję `DeleteObject(x);`. Użytkownik może łatwo pisać własne polityki zwalniania, a co za tym idzie używać tych inteligentnych wskaźników do przechowywania uchwytów na różnego rodzaju zasoby.

Poniższy listing przedstawia przykład użycia inteligenego wskaźnika:

```
{
  scoped_ptr<int, DeleteArrayPolicy> Ptr;
  Ptr.reset(new int[128]);
  UseArray(Ptr.get());
} // Tu tablica zostaje automatycznie zwolniona
```

Moduł posiada bogaty zbiór funkcji operujących na łańcuchach znaków typu `string`. Pośród tych prostych wymienić można np. sprawdzanie, czy znak jest cyfrą, literą, znakiem alfanumerycznym, zamianę tekstu na duże lub małe litery. Oprócz nich dostępne są także liczne bardziej zaawansowane funkcje, w tym:

- konwersja między rodzajami kodowania znaków końca wiersza (stosowany w Windows `CR+LF`, stosowany w Unix `LF`, stosowany w Mac `CR`),
- konwersja między sposobami kodowania polskich znaków (obsługiwane strony kodowe to: Windows–1250, ISO–8859–2, CP852, UTF–8),
- kodowanie Rot13,
- dopasowanie łańcucha do maski zawierającej symbole wieloznaczne (ang. *Wildcards*) `?` i `*`,
- przeszukiwanie pełnotekstowe, które zwraca wyliczoną trafność poszukiwania podanego podłańcucha w danym łańcuchu,

- obliczanie odległości edycyjnej między dwoma łańcuchami — *Levenshtein Distance*,
- funktor służący do sortowania łańcuchów w tzw. porządku naturalnym (gdzie liczby są interpretowane numerycznie, tak że `Ala3` będzie wcześniej niż `Ala22`),
- predykat porównujący łańcuchy bez rozróżniania wielkości liter.

Moduł zawiera też zbiór funkcji operujących na łańcuchach w postaci ścieżek systemu plików. Obsługiwane są prawidłowo zarówno ścieżki systemu Windows, jak i Linux. Na przykład funkcje `ExtractFilePath`, `ExtractFileName`, `ExtractFileExt` pozwalają na pobieranie konkretnego fragmentu ścieżki. Najbardziej zaawansowane w tej grupie są funkcje `RelativeToAbsolutePath` i `AbsoluteToRelativePath`, które służą do zamiany ścieżki względnej na bezwzględną i odwrotnie. Implementacja funkcji do operacji na ścieżkach inspirowana była w dużym stopniu podobnymi funkcjami w Delphi.

Autor zaimplementował konwersje między łańcuchami znaków, a wartościami różnych typów, przede wszystkim liczbowych. Decyzja o własnej implementacji podjęta została z powodu niezadowolenia funkcjami wbudowanymi w bibliotekę standardową C — przede wszystkim ich bezpieczeństwem w sensie nieumiejętności zgłaszania błędów przekroczenia zakresu czy nieprawidłowych znaków.

Konwersji między łańcuchami a typami całkowitoliczbowymi dowolnej długości dokonują szablony funkcji `UnitToStr` i `IntToStr` oraz `StrToUint` i `StrToInt`. Pozwalają one na podanie podstawy systemu, co umożliwia używanie zarówno systemu dziesiętnego, jak i szesnastkowego, ósemkowego, binarnego i innych. Ponadto dostępne są konwersje dla typów: zmiennoprzecinkowych `float` i `double`, znakowego `char`, logicznego `bool` i wskaźnikowego `const void*`. Dodatkowo funkcje `IntToStr2` i `UnitToStr2` potrafią dopełniać łańcuch do określonej długości zerami, a funkcja `SizeToStr` zwraca łańcuch z rozmiarem odpowiadającym podanej liczbie bajtów, wyrażony w automatycznie dostosowanych jednostkach (B, KB, MB, GB itd.). Przykład użycia:

```
uint4 i = 5;
string s;
UintToStr2(&s, i, 3); // s zawiera "005"
```

Na podstawie powyższych funkcji zbudowany został uogólniony, rozszerzalny mechanizm konwersji łańcuchów na różne typy i odwrotnie, oparty na specjalizacji szablonów funkcji `SthToStr` i `StrToSth`. Dzięki niemu konwersja przebiega tak samo niezależnie od typu. Przykład:

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
string s;
SthToStr(&s, v); // s zawiera "1,2,3"
```

Dalej na bazie tego mechanizmu powstał wygodny w użyciu mechanizm „formatowania” łańcuchów zawierających wartości pobierane z wielu zmiennych, wzorowany na analogicznym z Boost. Choć jego użycie nie jest najoptymalniejszym wyborem, działa dużo szybciej niż Boost Format (który używa powolnych strumieni biblioteki standardowej C++). Przykład użycia:

```
int i = 123;
float f = 3.14f;
string s = "abc";
string Out = Format("i=#, f=#, s=#") % i % f % s;
// Out zawiera "i=123, f=3.14, s=abc"
```

Bardzo ważny (np. w grach) jest precyzyjny pomiar czasu. Musi on być dużo dokładniejszy, niż co do sekundy, a najlepiej rzędu części milisekund. Moduł bazowy posiada przeznaczoną do tego klasę `TimeMeasurer`. Na platformie Windows używa ona licznika QPC (ang. *Query Performance Counter*) — funkcji `QueryPerformanceFrequency` i `QueryPerformanceCounter`, a na platformie Linux funkcji `gettimeofday`.

Ponadto dostępna jest funkcja `Wait` wstrzymująca bieżący wątek na określony czas (czekająca). Na platformie Windows używa funkcji systemowej `Sleep`, a na platformie Linux funkcji `select`.

Bardziej zaawansowane funkcje matematyczne zgromadzone są w module matematycznym — zobacz p. 2.3. W opisywanym tutaj module bazowym umieszczone zostały jedynie te, które nie używają specjalnych typów takich jak wektory, macierze czy kwaterniony.

W skład tej grupy wchodzą stałe matematyczne takie jak `PI_X_2` ($2\pi$), `LOG2E` ($\log_2 e$) czy `SQRT2` ($\sqrt{2}$). Proste funkcje *inline* uzupełniają braki biblioteki standardowej C i C++. Pośród nich znajdują się na przykład szablony funkcji `safe_add`, `safe_sub` i `safe_mul` służące do dodawania, odejmowania i mnożenia liczb całkowitych dowolnej długości bez obawy o przepełnienie zakresu, funkcja `round` zaokrąlająca liczbę zmiennoprzecinkową do całkowitej zgodnie z zasadami matematyki, funkcja `ceil_div` dzieląca dwie liczby całkowite z zaokrągleniem wyniku w górę, porównywanie liczb zmiennoprzecinkowych z podaną tolerancją, znajdowanie najmniejszej potęgi dwójki większej lub równej podanej liczbie, szybkie podnoszenie liczby do potęgi całkowitej, funkcje `trunc` i `frac` zwracające odpowiednio część całkowitą i część ułamkową podanej liczby zmiennoprzecinkowej, funkcja rozwiązująca równanie kwadratowe czy kilka funkcji do operacji na kątach.

Na szczególną uwagę zasługują bardziej złożone funkcje. `SmoothCD` służy do wygładzania zmian pewnej wielkości (np. w czasie) wg metody *Critically Damped Smoothing* [59]. Dostępne są też funkcje generujące 1–, 2– oraz 3–wymiarowy szum Perlina.

Dostępny jest także zbiór funkcji okresowych. Każdej z nich używa się w ten sam sposób — podając zmienną oraz parametry: wartość bazową, amplitudę, częstotliwość i fazę. Funkcje różnią się kształtem przebiegu. Dostępne są funkcje: sinusoidalna,

trójkątana, prostokątna, piłokształtna, odwrotna piłokształtna oraz prostokątna z regulowanym współczynnikiem wypełnienia. Funkcje takie są bardzo przydatne w programowaniu gier do obliczania zmian różnych wielkości w czasie (jak pozycja, orientacja, kolor, rozmiar, jasność, przezroczystość itd.).

Moduł dostarcza własną implementację generatora liczb pseudolosowych — klasę `RandomGenerator`. Tworzenie własnych obiektów tej klasy pozwala na posiadanie prywatnego generatora dla konkretnego procesu obliczeniowego czy w konkretnym wątku. Z kolei ręczne ustawianie ziarna umożliwia otrzymanie generatora deterministycznego. Domyślnie jako ziarno pobierany jest czas systemowy.

Generowaniu wysokiej jakości liczb pseudolosowych poświęcone zostało wiele publikacji. W programowaniu gier jednak na pierwszym miejscu stoi wydajność. Dlatego generowanie liczby typu `uint4` zrealizowane zostało za pomocą jednego dodawania i jednego mnożenia, a generowanie liczby zmiennoprzecinkowej `float` — za pomocą sztuczki opartej na budowie bitowej tego typu.

Generator zapewnia losowanie liczb całkowitych (z pełnego lub z podanego zakresu), zmiennoprzecinkowych (z zakresu podanego lub domyślnego $[0.0; 1.0]$), wartości logicznych, dowolnie długich danych binarnych oraz liczb zmiennoprzecinkowych wg rozkładu normalnego (Gaussa).

Szablon klasy `Singleton` ułatwia pisanie klas wg wzorca projektowego *Singleton*, czyli takich, które posiadają co najwyżej jedną instancję tworzoną w chwili pierwszego użycia. Korzysta z możliwości sparametryzowania szablonu klasą, która z niego dziedziczy. Dostarcza prywatnego pola przechowującego utworzoną instancję oraz metody statycznej `GetInstance` do jej pobierania. Przykład użycia:

```
class MyClass : public Singleton<MyClass>
{
public:
  void Foo();
};

...

MyClass::GetInstance().Foo(); // Obiekt powstaje przy pierwszym użyciu
MyClass::GetInstance().Foo(); // Obiekt już istnieje
```

Interpretacja przełączników podanych jako parametry wiersza poleceń podczas uruchamiania programu nie jest wbrew pozorom prostym zadaniem. W aplikacjach okienkowych w Windows cały wiersz poleceń ma postać pojedynczego łańcucha przekazywanego do funkcji `WinMain`. W aplikacjach konsolowych i w Linux wiersz poleceń jest rozbity na poszczególne parametry przekazywane w tablicy łańcuchów do funkcji `main`. Nadal jednak wyzwaniem pozostaje rozłożenie tych parametrów na poszczególne przełączniki. W Linuksie dostępna jest przeznaczona do tego funkcja `getopt`, nie ma jej jednak w Windowsie.

Dlatego po dogłębnych badaniach zachowania się wiersza poleceń w obydwu systemach autor zdecydował się na napisanie własnego parsera parametrów. Ma on

postać klasy `CmdLineParser`. Jako wejście akceptuje ona zarówno pojedynczy łańcuch z całym wierszem poleceń (`WinMain`), jak i tablicę parametrów (`main`). Obsługuje przełączniki zapisane zarówno w konwencji Windowsa (`/opcja`), jak i Linuksa (`-o` `--opcja`).

Użycie parsera wymaga najpierw zarejestrowania wszystkich dostępnych przełączników. Każdy z nich może być pojedynczym znakiem lub dłuższym łańcuchem i każdy ma określoną flagę, czy przyjmuje dodatkowy parametr. Potem można rozpocząć odczytywanie kolejnych elementów. Przełączniki jednoznakowe można łączyć, np. zamiast `-a -b -c` napisać można `-abc`. Dodatkowy parametr można zapisywać na różne sposoby: `-a TEKST, -aTEKST, -a=TEKST`.

Oto przykład. Po zarejestrowaniu następujących przełączników:

```
RegisterOpt(1, ’a’, false);
RegisterOpt(2, ’b’, false);
RegisterOpt(3, ’c’, true);
RegisterOpt(11, "AA", false);
RegisterOpt(12, "BBB", true);
```

parsować można następujący wiersz poleceń:

```
-a -b -c param -abc="param" "-cparam" /AA --AA "/BBB"=param TEKST
  --BBB "param"
```

**Moduł Config** W wielu różnego rodzaju programach zachodzi potrzeba wczytywania, a czasami też zapisywania ustawień konfiguracyjnych w plikach. Najczęściej są to pliki tekstowe, których zaletą jest czytelność i możliwość ręcznej edycji przez użytkownika za pomocą zwykłego edytora tekstu.

Moduł do obsługi konfiguracji to kod zgromadzony w plikach `Common\Config.hpp` i `Common\Config.cpp`. Stanowi go zbiór struktur, które pozwalają na budowanie w pamięci reprezentacji pliku konfiguracyjnego i względnie szybki oraz wygodny dostęp do przechowywanych przez niego danych. Zdefiniowane są trzy rodzaje struktur jako podklasy dziedziczące z klasy bazowej `Item`:

- Klasa `Value` przechowuje pojedynczą wartość (zawsze typu łańcuchowego, ale może być konwertowana na różne typy danych — p. niżej).
- Klasa `List` zawiera uporządkowaną sekwencję wartości typu łańcuchowego zapamiętaną w wektorze STL.
- Klasa `Config` zawiera mapę przechowującą zbiór par „Klucz → Wartość”, gdzie klucz jest łańcuchem znaków, a wartość jest obiektem dowolnego z tych trzech typów.

Dzięki temu w pamięci reprezentowane może być całe drzewo danych odwzorowujące plik konfiguracyjny. Klasa `Config` reprezentuje zarówno konfigurację jako całość, jak i dowolną jej gałąź. Posiada ona metody pozwalające na wczytywanie i zapisywanie konfiguracji do/z pliku, łańcucha lub dowolnego strumienia, a także do wygodnego

dostępu do zgromadzonych wartości za pomocą „ścieżki" wyrażonej jako łańcuch w postaci `Konfiguracja/PodKonfiguracja/Element`. Ponadto jej metody pozwalają na automatyczną konwersję pamiętanych wartości łańcuchowych z/do wartości dowolnych typów wspieranych przez mechanizm `StrToSth` i `SthToStr` opisany w rozdziale 2.3. Prosty przykład użycia:

```
common::Config cfg;
cfg.LoadFromFile("Settings.cfg");

int Timeout;
cfg.MustGetDataEx("GeneralSettings/Timeout", &Timeout);
```

**Moduł DateTime** Obsługa daty i czasu pojmowanego w taki sposób, w jaki używają ich ludzie zgodnie z kalendarzem gregoriańskim jest dla programisty nie lada wyzwaniem ze względu na złożoność tego kalendarza. Na przykład w celu obliczenia, jaki dzień tygodnia przypada dla określonej daty, algorytm polega najpierw na przeliczeniu tej daty na datę juliańską (JDN — ang. *Julian Day Number*).

Istnieje wiele dostępnych API do obsługi daty i czasu. Na przykład biblioteka standardowa C definiuje typy takie, jak `struct tm`, `clock_t`, `time_t`, WinAPI używa do reprezentowania czasu typów `SYSTEMTIME`, `FILETIME` i `DWORD`. Moduł do obsługi daty i czasu dostarcza też biblioteka Boost. Autor zdecydował się na zaimplementowanie własnej biblioteki tego typu. Implementacja zaprezentowana tutaj wzorowana jest na module datetime z biblioteki wxWidgets. Stanowi ją kod zgromadzony w plikach `Common\DateTime.hpp` i `Common\DateTime.cpp`.

Moduł definiuje szereg typów reprezentujących różnego rodzaju wartości związane z datą i czasem. Dostępne są konwersje między tymi typami, jak również konwersje tych typów do/z łańcuchów znakowych. Wiele z nich ma także przeładowane operatory pozwalając na intuicyjne dokonywanie wybranych operacji arytmetycznych (jak dodawanie, odejmowanie, mnożenie, porównania).

- `WEEKDAY` to typ wyliczeniowy reprezentujący dzień tygodnia.
- `MONTH` to typ wyliczeniowy reprezentujący miesiąc roku.
- `NAME_FORM` to flagi bitowe reprezentujące sposób generowania nazw dla dni tygodnia i miesięcy
- `DATESPAN` reprezentuje odcinek czasu. Przechowuje osobne liczby całkowite dla lat, miesięcy, tygodni i dni. Można za jego pomocą przesuwać `TMSTRUCT`.
- `TIMESPAN` reprezentuje odcinek czasu. Przechowuje liczbę całkowitą ze znakiem w milisekundach. Można za jego pomocą przesuwać `DATETIME`.
- `TMSTRUCT` reprezentuje moment czasu. Przechowuje osobne pola: dzień, miesiąc, rok, godzina, minuta, sekunda, milisekunda. Pamięta także dzień tygodnia, który wyliczany jest przy pierwszym odczytaniu (pole `mutable`).
- `DATETIME` reprezentuje moment czasu. Przechowuje liczbę milisekund od „epoki uniksowej", czyli 1 stycznia 1970.

**Moduł Dator** Moduł ten stanowi kod zgromadzony w plikach `Common\Dator.hpp` i `Common\Dator.cpp`. „Dator” to obiekt otaczający pojedynczą wartość jakiegoś typu i dający dostęp do jej zapisu i odczytu poprzez łańcuch tekstowy. Przechowuje wskaźnik do tej wartości. Dator działa dla wszystkich typów obsługiwanych przez `SthToStr` i `StrToSth` (patrz p. 2.3). Można też tworzyć własne specjalizacje. Wszystkie datory parametryzowane różnymi typami mają wspólną klasę bazową `GenericDator` z wirtualnymi metodami `SetValue` i `GetValue`.

Ponadto dostępna jest klasa `DatorGroup`, która przechowuje w swoim wnętrzu i zarządza kolekcją dowolnego rodzaju datorów identyfikowanych po nazwach. Może ona stanowić bazę dla utworzenia prostej tablicy właściwości (ang. *Property Grid*) — kontrolki dającej jednolity dostęp do kolekcji wartości różnego typu, odgrywającej coraz większą rolę w różnego rodzaju edytorach graficznych, również w branży gier komputerowych. Przykład użycia:

```
int I = 123;
float F = 10.5f;
common::DatorGroup DG;
DG.Add("Strength", &I);
DG.Add("Life", &F);

DG.SetValue("Strength", "124"); // Teraz I wynosi 124.
string s;
DG.GetValue("Life", &s); // Teraz s wynosi "10.5".
```

**Moduł Error** Moduł ten stanowi kod zgromadzony w plikach `Common\Error.hpp` i `Common\Error.cpp`. Składa się na niego hierarchia klas przeznaczonych do użycia jako wyjątki C++. Zapewnia w ten sposób ujednolicony system zgłaszania błędów w kodzie, na który należy „przetwarzać” błędy raportowane na różne sposoby przez różne używane biblioteki. Modułu tego używają też inne moduły biblioteki bazowej (jak `Files` czy `Stream`) oraz pozostała część opisywanego w tej pracy kodu.

W grach komputerowych, w przeciwieństwie do innego typu aplikacji, błędy nigdy nie powinny wystąpić. Jeśli błąd wystąpi, program najczęściej kończy swoje działanie. Ponadto obsługa błędów w C++ spowalnia wykonywanie kodu, co ma kluczowe znaczenie w tej dziedzinie. Dlatego profesjonaliści z tej branży często całkowicie wyłączają obsługę wyjątków C++ w opcjach kompilatora.

Z drugiej jednak strony, dobre raportowanie błędów ma bardzo duże znaczenie na etapie powstawania kodu. Na przykład użycie wartości niezainicjalizowanej (gdyż nie udało się jej wczytać z pliku konfiguracyjnego) owocuje dziwnym i nieprzewidywalnym zachowaniem programu, a próba użycia wskaźnika do obiektu, którego nie udało się utworzyć, zakończy się błędem ochrony pamięci. Takie błędy dużo trudniej jest zidentyfikować i naprawić, niż gdyby kod każdorazowo sprawdzał powodzenie operacji (szczególnie takich jak wszelkie wejście–wyjście i tworzenie różnego rodzaju zasobów) oraz raportował ewentualne błędy w czytelny sposób. Dlatego autor zdecydował się jednak na użycie wyjątków w swoim kodzie.

Powstaje pytanie, jakie informacje powinien nieść wyjątek? Wykonywanie wszelkiego kodu opiera się na stosie wywołań i rodzaj błędu możnaby raportować bądź to z miejsca, w którym powstaje (najniższy poziom), bądź też z miejsca, w którym zostaje przechwycony (najwyższy poziom). Zdaniem autora żadne z tych podejść nie dostarcza wystarczająco dużo informacji potrzebnych do zidentyfikowania przyczyny błędu. Jeśli na przykład błąd powstaje w kodzie wczytującym plik konfiguracyjny, zarówno komunikat najniższego poziomu — „Nie można skonwertować łańcucha na liczbę", jak i komunikat najwyższego poziomu — „Nie można uruchomić programu" — nie daje pełnego obrazu zaistniałej sytuacji. Dostarczyć go może dopiero w miarę kompletny stos wywołań, jak na przykład:

```
Nie można uruchomić programu.
Nie można wczytać pliku konfiguracyjnego "Config01.cfg".
Błąd składni: wiersz 12, kolumna 3.
Nie można skonwertować łańcucha na liczbę.
```

Istnieją mechanizmy pozwalające na otrzymanie stosu wywołań funkcji od systemu. Same nazwy funkcji nie są jednak wystarczające, bo podczas odwijania stosu przy zgłaszaniu błędu konieczne jest niejednokrotnie dodawanie dodatkowych informacji, takich jak nazwa pliku, numer wiersza i kolumny w tekście i inne dane kluczowe dla zidentyfikowania miejsca, w którym wystąpił błąd. Dlatego autor zdecydował się umieścić w klasie bazowej wyjątku `Error` cały stos łańcuchów znakowych niosących komunikaty o przyczynach błędu.

Oprócz możliwości zgłoszenia błędu poprzez rzucenie wyjątku z utworzonego obiektu odpowiedniej klasy, potrzebna jest możliwość dopisania nowych komunikatów podczas odwijania stosu. Można to zrobić za pomocą słowa kluczowego `catch` przyjmującego referencję do obiektu wyjątku, dopisania komunikatu do tego obiektu i jego dalszego rzucenia instrukcją `throw;`. Aby uprościć to zadanie, moduł posiada zdefiniowane makra, w tym `ERR_TRY`, `ERR_CATCH`, `ERR_CATCH_FUNC`.

W językach gdzie używanie wyjątków do obsługi błędów jest standardem (np. Java, C#), hierarchia klas wyjątków odzwierciedla zwykle rodzaje błędów. W niniejszym kodzie jednak nie ma potrzeby wyłapywania tylko wyjątków wybranego rodzaju, ponieważ prawie wszystkie błędy są obsługiwane na najwyższym poziomie i powodują zakończenie programu. Dlatego dziedziczenie zostało wykorzystane do wyprowadzenia klas wyjątków różniących się sposobem powstania. Są to klasy niepolimorficzne, a ich jedynym zadaniem jest dostarczenie konstruktora, który potrafi przetworzyć błąd zgłaszany na sposób charakterystyczny dla danej biblioteki na jednolitą reprezentację za pomocą wyjątków opisywanego modułu. I tak, klasa bazowa `Error` zapewnia zgłaszanie błędów własnych, podczas gdy na przykład klasa `DirectXError` przyjmuje w konstruktorze dodatkowo kod błędu typu `HRESULT` zwrócony przez funkcję DirectX i automatycznie pobiera komunikat błędu na podstawie tego kodu. Analogicznie działają inne klasy wyjątków dostosowane do sposobu zgłaszania błędów przez różne biblioteki:

1. `Error` — klasa bazowa przeznaczona do zgłaszania błędów własnych.
2. `ErrnoError` — klasa przeznaczona do zgłaszania błędów na podstawie kodu `errno`, używanego przez funkcje biblioteki standardowej C i API systemu Linux. Automatycznie pobiera kod i komunikat błędu ze zmiennej `errno`.
3. `Win32Error` — klasa przeznaczona do zgłaszania błędów biblioteki Windows API. Automatycznie pobiera kod i komunikat błędu z funkcji `GetLastError`.
4. `SDLError` — klasa przeznaczona do zgłaszania błędów biblioteki SDL.
5. `OpenGLError` — klasa przeznaczona do zgłaszania błędów biblioteki OpenGL.
6. `FmodError` — klasa przeznaczona do zgłaszania błędów biblioteki dźwiękowej FMOD.
7. `DirectXError` — klasa przeznaczona do zgłaszania błędów biblioteki DirectX.
8. `WinSockError` — klasa przeznaczona do zgłaszania błędów biblioteki WinSock (będącej częścią Windows API).
9. `DevILError` klasa przeznaczona do zgłaszania błędów biblioteki DevIL (przeznaczonej do obsługi różnych graficznych formatów pliku).
10. `AVIFileError` — klasa przeznaczona do zgłaszania błędów biblioteki AVI File (będącej częścią Windows API).

Przykład:

```
void RobCos()
{
  ERR_TRY;
  throw common::Error("Jakiś błąd");
  ERR_CATCH_FUNC;
}

void WczytajKonfiguracje(const string &NazwaPliku)
{
  ERR_TRY;
  RobCos();
  ERR_CATCH("Nie można wczytać pliku: " + NazwaPliku);
}

void UruchomProgram()
{
  ERR_TRY;
  WczytajKonfiguracje("Config.dat");
  ERR_CATCH("Nie można uruchomić programu");
}

void Test()
{
  try
  {
    UruchomProgram();
  }
  catch (const common::Error &e)
  {
    string Message;
    e.GetMessage_(&Message);
    MessageBox(g_MainWnd, Message.c_str(), "Mój program",
      MB_OK | MB_ICONERROR);
  }
}
```

Powyższy przykład pokaże komunikat błędu podobny do tego:

```
[test.cpp,1611] Nie można uruchomić programu
[test.cpp,1602] Nie można wczytać pliku: Config.dat
[test.cpp,1593] void __cdecl RobCos(void)
Jakiś błąd
```

**Moduł Files** Kod tego modułu zgromadzony jest w plikach `Common\Files.hpp` i `Common\Files.cpp`. Zapewnia przenośne (jak cała biblioteka bazowa) funkcje do obsługi systemu plików, a więc plików i katalogów.

Klasa `FileStream` rozszerza hierarchię klas strumieni (zobacz p. 2.3) o strumień służacy do obsługi plików dyskowych. Klasa `DirLister` służy do listowania zawartości wybranego katalogu — leżących w nim plików i podkatalogów.

Ponadto moduł dostarcza funkcji służących m.in. do zapisywania i odczytywania plików w całości (jako surowe dane binarne lub łańcuchy znaków), do pobierania i ustawiania właściwości plików (rozmiar, data i czas utworzenia, ostatniej modyfikacji i ostatniego dostępu), a także do tworzenia i usuwania plików oraz katalogów.

Większość z tych funkcji posiada dwie wersje. Pierwsza raportuje powodzenie zwracając wartość typu `bool`, natomiast druga (której nazwa rozpoczyna się od `Must`) w przypadku niepowodzenia rzuca wyjątek.

W celu zapewnienia jak najlepszej wydajności oraz jakości raportowanych błędów, moduł używa funkcji z biblioteki standardowej C, ale na platformie Windows tam gdzie to możliwe preferuje używanie funkcji WinAPI.

**Moduł FreeList** Dynamiczna alokacja pamięci ze sterty nie należy do szybkich operacji. Oczywiście jest wielokrotnie szybsza niz na przykład wejście–wyjście do plików dyskowych. Jednak dokonywanie wielu alokacji w każdej klatce silnika może znacząco spowolnić cały program.

Dlatego wielu programistów gier decyduje się na własną implementację alokacji pamięci. Jest to powszechne szczególnie na konsolach. Na platformie PC, ze względu na dużą ilość dostępnej pamięci i obecność pliku wymiany nie jest to aż tak potrzebne.

Sytuacją, kiedy alokacja pamięci najbardziej spowalnia proces obliczeniowy jest tworzenie wielu małych obiektów — np. cząsteczek dla efektu cząsteczkowego czy węzłów drzewa dla struktury podziału przestrzeni. Tak się jednak składa, że alokację obiektów określonego typu, których rozmiar w pamięci jest stały i znany, można łatwo przyspieszyć stosując technikę listy wolnych elementów (ang. *Free List*) [60].

Technika ta polega na zaalokowaniu większego bloku pamięci zdolnego pomieścić wiele elementów i przydzielaniu pamięci z fragmentów tego bloku. Wolne elementy tworzą listę łączoną jednokierunkową, dzięki której alokacja i zwalnianie odbywa się w czasie stałym $O(1)$. Sztuczką optymalizacyjną jest wykorzystanie pierwszych czterech bajtów wolnego elementu do pamiętania wskaźnika na następny wolny element, zamiast tworzenia osobnej listy. Od strony języka C++ implementację tej techniki,

działającą także dla własnych nietrywialnych klas, można wykonać używając operatora *placement new* oraz jawnego wywołania destruktora.

Zalety alokatora opartego na *Free List* to szybsza alokacja i zwalnianie pamięci oraz lepsza lokalność odwołań, dodatkowo poprawiająca wydajność dzięki lepszemu wykorzystaniu pamięci podręcznej *Cache*. Wady natomiast, to konieczność utworzenia obiektu *Free List*, alokacja więcej pamięci niż to jest potrzebne (część pamięci się marnuje), niestandardowy sposób alokowania i zwalniania obiektów oraz, w przypadku typowej prostej implementacji, ograniczona z góry maksymalna liczba możliwych do zaalokowania elementów.

Wszystkie opisane powyżej rozwiązania wykorzystane zostały w niniejszym module. Kod tego modułu znajduje się w pliku `Common\FreeList.hpp`. Dostarcza on dwie klasy. `FreeList` to podstawowy alokator, rezerwujący pamięć na z góry określoną liczbę elementów. `DynamicFreeList` działa nieco wolniej, ale potrafi rezerwować wiele bloków pamięci, dzięki czemu liczba elementów możliwych do zaalokowania za jego pomocą ograniczona jest wyłącznie przez system operacyjny. Oto przykłady użycia:

```
// Rezerwuje blok 1024 elementów.
common::FreeList<int> L1(1024);
// Alokuje element bez jego inicjalizacji
int *Ptr1 = L1.New();
// Alokuje element inicjalizując go konstruktorem int(), czyli zerem.
int *Ptr2 = L1.New_ctor();
// Zwalnia zaalokowane elementy
L1.Delete(Ptr2);
L1.Delete(Ptr1);

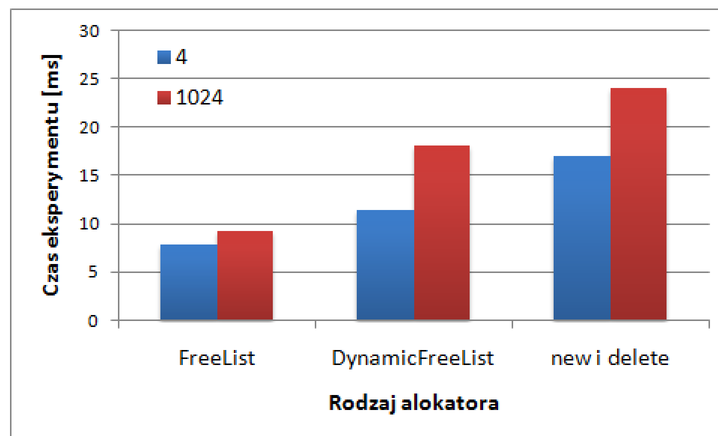
// Dynamiczna lista o rozmiarze pojedynczego bloku 128 elementów.
common::DynamicFreeList<MyClass> L2(128);
// Alokacja obiektu z wywołaniem konstruktora dwuargumentowego
MyClass *Obj = L2.New(1, 2);
// Zwolnienie obiektu z wywołaniem destruktora
L2.Delete(Obj);
```

W celu empirycznego upewnienia się o zasadności stosowania tego modułu, autor przeprowadził eksperyment polegający na pomiarze czasu trwania następującego testu: Każdy test składał się z 10240 losowych operacji, z 90% szansy, że tą operacją będzie alokacja nowego elementu i 10% szansy, że tą operacją będzie zwolnienie jednego z zaalokowanych elementów. Dodatkowo na końcu następowało zwolnienie wszystkich zaalokowanych i niezwolnionych wcześniej elementów. Wyniki pokazuje tabela 2.2 oraz wykres 2.2.

**Moduł Logger** Mianem **loga** (dziennika) określa się lokalizację (najczęściej plik tekstowy), do której program zapisuje komunikaty informujące o jego stanie pracy, ważnych zdarzeniach, błędach itd. Logowania używają różnego typu programy, szczególnie te nieposiadające okienkowego interfejsu użytkownika — np. aplikacje serwerowe oraz gry i silniki. Autor projektując opisany tutaj logger starał się, aby był on jak najbardziej uniwersalny, ogólny i nadawał się dobrze do wszystkich tych zastosowań.

Tablica 2.2. Wyniki pomiaru wydajności alokatorów modułu `FreeList` w porównaniu z domyślnym alokatorem systemu Windows.

| Konfiguracja | Rozmiar elementu | `FreeList` | `DynamicFreeList` | `new` i `delete` |
|---|---|---|---|---|
| DEBUG | 4 B | 68.0636 ms | 184.441 ms | 78.8142 ms |
| DEBUG | 1024 B | 69.3896 ms | 203.506 ms | 93.2942 ms |
| RELEASE | 4 B | 7.8722 ms | 11.4786 ms | 17.0348 ms |
| RELEASE | 1024 B | 9.1805 ms | 18.0729 ms | 24.0537 ms |




Rys. 2.2. Czas trwania eksperymentu alokacji pamięci różnymi metodami w konfiguracji RELEASE.

Kod tego modułu zgromadzony jest w plikach `Common\Logger.hpp` i `Common\Logger.cpp`. W opisywanym module, **logger** jako cały system logowania jest zawsze jeden. Inicjalizuje się go funkcją `CreateLogger`, a finalizuje funkcją `DestroyLogger`. Można w nim rejestrować wiele obiektów logów, które będą zapisywały komunikaty do różnych lokalizacji. Można do niego zapisywać też różne rodzaje komunikatów. Głównym założeniem jest przy tym, że podział komunikatów na rodzaje jest zupełnie oddzielony od logów, do jakich mogą zostać wysłane.

Obiekty logów reprezentują konkretne lokalizacje, do których mogą być zapisywane komunikaty. Można definiować własne klasy logów dziedzicząc z klasy bazowej `ILog`. Dostępne są predefiniowane typy: `TextFileLog` zapisuje komunikaty do pliku tekstowego. `HtmlFileLog` zapisuje komunikaty do pliku HTML. `OstreamLog` zapisuje komunikaty do strumienia biblioteki standardowej C++ (np. na konsolę systemową `std::cout`).

Komunikat powinien mieć, oprócz treści, również pewien określony **typ**. Może on reprezentować np. rodzaj komunikatu (diagnostyczny, debugowy, informacyjny), jego priorytet (informacja, ostrzeżenie, błąd) czy rodzaj nadawcy (od którego podsystemu pochodzi). Jako najbardziej ogólne rozwiązanie autor wyposażył zapisywany komunikat w liczbę całkowitą 32-bitową (`unit4`), której konkretna interpretacja zależy od użytkownika tej biblioteki. Liczba ta powinna być traktowana jako wzorzec bitowy, gdyż w loggerze określać można maski bitowe, na podstawie których komunikaty wysyłane są do zera, jednego lub wielu spośród zarejestrowanych w nim logów.

Owa maska bitowa może być wykorzystana dodatkowo do poprzedzania wybranych rodzajów komunikatów określonym **prefiksem**, który jest dodawany na początku treści komunikatu. Prefiks taki to dowolny ciąg znaków, a specjalne sekwencje są w nim zastępowane aktualną datą, czasem i dodatkowymi informacjami ustawianymi w loggerze (silnik wykorzystuje je do podania bieżącego numeru klatki). Istnieje osobno jeden główny prefiks dodawany do wszystkich komunikatów i osobno lista prefiksów z przypisanymi maskami bitowymi dla określonych rodzajów komunikatów. Ponadto prefiksy można ustalać dla każdego loga osobno lub globalnie dla całego loggera, co powoduje ich dodanie w każdym z istniejących logów. Oprócz prefiksów tekstowych dodawanych na początku treści komunikatów, pewne rodzaje logów mogą posiadać własne mapowanie masek bitowych na dodatkowe parametry takie jak kolor czy pogrubienie danego komunikatu, jeśli takie formatowania są dostępne.

Logger może pracować w sposób bardziej bezpieczny lub bardziej wydajny. W celu osiągnięcia najwyższej wydajności można włączyć tryb pracy z kolejką, w którym logger pracuje **w osobnym wątku** logując zakolejkowane komunikaty w swoim tempie. Ponadto pliki logów mogą pozostawać otwarte przez cały czas działania programu. Jednakże jeśli log ma za zadanie nieść informacje krytyczne dla zidentyfikowania przyczyny błędu i ostatnia taka informacja może się pojawić na chwilę przez awarią programu (np. błąd ochrony pamięci), konieczne jest większe bezpieczeństwo, aby mieć pewność, że komunikat zostanie zalogowany zanim system operacyjny zamknie proces. Wówczas należy nie włączać trybu pracy z kolejką, a dla logów używających plików wybrać tryb, w którym po zalogowaniu każdego komunikatu następuje *Flush* lub nawet osobne otwarcie pliku.

Poniższy kod ilustruje niektóre możliwości opisanego loggera:

```
// Inicjalizacja loggera
CreateLogger(false);
// Pobranie obiektu loggera
Logger & L = GetLogger();

// Utworzenie przykładowych logów
TextFileLog *Log1 = new TextFileLog("Log1.txt", FILE_MODE_NORMAL,
  EOL_CRLF);
TextFileLog *Log2 = new TextFileLog("Log2.txt", FILE_MODE_NORMAL,
  EOL_CRLF);

// Ustawienie masek bitowych dla poszczególnych logów
L.AddLogMapping(0xFFFFFFFF, Log1);
L.AddLogMapping(1, Log2);

// Ustawienie prefiksów
Log1->SetPrefixFormat("[%D %T %1] ");
Log1->AddTypePrefixMapping(1, "(!) ");
L.SetCustomPrefixInfo(0, "Frame:123");

// Przykładowe logowanie komunikatów
L.Log(1, "Komunikat 1"); // Trafi do Log1 i Log2
L.Log(2, "Komunikat 2"); // Trafi do Log1
L.Log(3, "Komunikat 3"); // Trafi do Log1 i Log2
```

```
// Finalizacja loggera
DestroyLogger();
// Zwolnienie logów
delete Log2;
delete Log1;
```

Komunikaty zapisane przez `Log1` będą wyglądały mniej więcej tak:

```
[2006-08-18 21:52:28 Frame:123] (!) Komunikat 1
[2006-08-18 21:52:28 Frame:123] Komunikat 2
[2006-08-18 21:52:28 Frame:123] (!) Komunikat 3
```

**Moduł Math** Kod tego modułu zgromadzony jest w plikach `Common\Math.hpp` i `Common\Math.cpp`. Jest to największy z modułów biblioteki bazowej. Dostarcza on typów i funkcji związanych z geometrią i stanowi podstawę dla wszelkich obliczeń w całym opisywanym systemie. Należy przy tym podkreślić, że moduł ten zawiera wyłącznie elementy przydatne w programowaniu gier i grafiki 3D, a nie jest ogólną biblioteką matematyczną. Różni się więc znacznie od bibliotek stosowanych np. w obliczeniach naukowych. Przykładowo, nie zawiera on klasy macierzy o dowolnym rozmiarze, a jedynie macierz $4 \times 4$, ponieważ takie właśnie macierze są stosowane do reprezentowania transformacji w przestrzeni 3D.

Użytkownicy DirectX są w o tyle komfortowej sytuacji w porównaniu z użytkownikami OpenGL, że na wyposażeniu tej pierwszej — w D3DX — jest już rozbudowana biblioteka matematyczna. Opisany tu moduł w znacznym stopniu dubluje jej funkcjonalność. Decyzja o napisaniu własnej implementacji podstawowych typów i funkcji zamiast skorzystania z oferowanych przez D3DX podyktowana była chęcią zapewnienia przenośności na platformę Linux, w której DirectX nie jest dostępny. Intencją autora była możliwość napisania w przyszłości aplikacji serwerowej, która musiałaby dokonywać obliczeń geometrycznych, a równocześnie pracowałaby w systemie uniksowym nie mając dostępu do biblioteki DirectX. Zdefiniowane struktury są jednak zbudowane tak samo jak te z D3DX, a więc są z nimi binarnie kompatybilne.

Kod tego modułu pisany był ze szczególną dbałością o wydajność. Nie używa on wyjątków modułu Error. Wszelkie dane o rozmiarze większym niż 4 bajty są zwracane przez funkcje za pomocą parametru wskaźnikowego, nie przez wartość. Jako liczby zmiennoprzecinkowe używany jest typ 32–bitowy pojedynczej precyzji — `float`.

Podstawą modułu matematycznego są różne obiekty geometryczne, z których większość posiada swoją reprezentację jako osobne struktury. Dla każdej z tych struktur zdefiniowany jest szereg przeładowanych operatorów, metod oraz funkcji globalnych, które nie są tutaj szczegółowo opisane. Każdy z nich daje się również skonwertować do/z łańcucha znaków za pomocą ujednoliconego mechanizmu `SthToStr` i `StrToSth` (p. rozdz. 2.3).

- **Punkt** 2D opisany liczbami całkowitymi reprezentuje struktura `POINT_`. Odpowiada ona typowi `POINT` z WinAPI i złożona jest z pól `int x, y;`.

- **Punkt** 2–, 3– i 4—wymiarowy opisany liczbami zmiennoprzecinkowymi reprezentują struktury odpowiednio `VEC2`, `VEC3` i `VEC4`. Posiadają one pola `float x, y;` oraz dodatkowo `z` i `w`. Odpowiadają typom z D3DX: `D3DXVECTOR2`, `D3DXVECTOR3` i `D3DXVECTOR4`. Zdefiniowane jest wiele funkcji do operacji na tych wektorach, w tym iloczyn skalarny i wektorowy, obliczanie długości i odległości, normalizacja, interpolacja liniowa, rzutowanie, ortogonalizacja i inne.
- **Trójkąt** w przestrzeni 3D nie posiada jawnej reprezentacji w postaci własnej struktury. Do jego opisania używane są trzy punkty typu `VEC3`. Zbiór funkcji wspierających trójkąty obejmuje m.in. operacje na współrzędnych barycentrycznych.
- **Promień** (ang. *Ray*, inaczej półprosta w przestrzeni 3D) również nie posiada swojej reprezentacji jako osobnej struktury. Do jego opisania używane są dwie zmienne typu `VEC3` — punkt początkowy oraz wektor kierunku (nazywane w kodzie `RayOrig` i `RayDir`, od ang. *Origin* i *Direction*).
- **Prostokąt** 2D zbudowany z liczb całkowitych reprezentuje struktura `RECTI`. Posiada ona pola `int left, top, right, bottom;`. Odpowiada strukturze `RECT` z WinAPI.
- **Prostokąt** 2D zbudowany z liczb zmiennoprzecinkowych reprezentuje struktura `RECTF`. Posiada ona pola `float left, top, right, bottom;`. Istnieją funkcje testujące czy punkt lub inny prostokąt zawiera się w danym prostokącie, jak również liczące sumę i część wspólną dwóch prostokątów.
- **AABB** (ang. *Axis–Aligned Bounding Box*, czyli prostopadłościan o krawędziach równoległych do osi układu współrzędnych) posiada reprezentację w postaci struktury `BOX` wyposażonej w pola opisujące jego dwa wierzchołki — o najmniejszych i o największych współrzędnych — `VEC3 p1, p2;`.
- **Kolor** wraz z czwartym kanałem — Alfa oznaczającym przezroczystość — posiada dwie reprezentacje. Struktura `COLOR` zbudowana jest z pojedynczej liczby całkowitej 4–bajtowej, której poszczególne bajty przechowują wartości czterech kanałów w zakresie $[0; 255]$. Struktura `COLORF` posiada pola `float R, G, B, A;` reprezentujące poszczególne składowe jako liczby zmiennoprzecinkowe w zakresie $[0.0; 1.0]$. Funkcje operujące na kolorach obejmują m.in. interpolację i konwersję do/z przestrzeni kolorystycznej HSB.
- **Płaszczyzna** w 3D opisywana jest równaniem $Ax+By+Cz+D = 0$. Jej współczynniki `float a, b, c, d;` potrafi przechowywać struktura `PLANE`. Odpowiada ona typowi `D3DXPLANE` z D3DX. Tworzenie płaszczyzny polega zazwyczaj na podaniu wektora normalnego i przykładowego punktu, który do niej należy bądź podaniu trzech należących do niej punktów.
- **Prosta** na płaszczyźnie opisywana jest równaniem $Ax + By + C = 0$. Do reprezentacji jej współczynników `float a, b, c;` zdefiniowana została struktura `LINE2D`.

- **Macierz** $4 \times 4$ opisuje struktura `MATRIX`. Używana jest w geometrii obliczeniowej do reprezentowania szerokiej klasy przekształceń w przestrzeni 3D. Jej podmacierz $3 \times 3$ potrafi reprezentować przekształcenia liniowe — dowolną rotację, skalowanie, ścinanie, odbicie oraz dowolne złożenie tych przekształceń w określonej kolejności. Macierz $4 \times 3$ potrafi reprezentować przekształcenia afiniczne — te wymienione powyżej oraz translację. W macierzy $4 \times 4$, dzięki wykorzystaniu współrzędnych jednorodnych dodatkowo reprezentowane może być rzutowanie perspektywiczne, co wyczerpuje większość przekszałceń stosowanych w grafice 3D (niemożliwe do opisania taką macierzą pozostają np. przekształcenia sferyczne i paraboloidalne). Liczne funkcje związane z tą strukturą służą przede wszystkim do tworzenia macierzy reprezentujących poszczególne rodzaje przekształceń.
- **Kwaternion** to rozszerzenie liczb zespolonych na wartość posiadającą 4 składowe — `float x, y, z, w;`. Nie powinien być mylony z punktem we współrzędnych jednorodnych. Dlatego reprezentowania kwaternionów utworzona została osobna struktura — `QUATERNION`. Odpowiada ona typowi `D3DXQUATERNION` z D3DX. Kwaternion wykorzystywany jest w grafice 3D do opisywania dowolnej rotacji lub orientacji i jest w tym zastosowaniu lepszy niż kąty Eulera czy macierz, ponieważ nie posiada zjawiska *Gimbal lock* oraz daje się łatwo interpolować (SLERP — ang. *Spherical Linear Interpolation*). Kwaternion powstaje przez podanie wektora osi obrotu i kąta obrotu wokół tej osi. Możliwe jest jednak dowolne przechodzenie między opisem rotacji przez kąty Eulera, macierz rotacji lub kwaternion. Do wszystkich tych przekształceń dostarczone są odpowiednie funkcje.
- **Frustum** to ścięty ostrosłup o podstawie prostokąta. Nie posiada eleganckiego polskiego tłumaczenia (w środowisku internetowym zaproponowany został termin „ściętosłup"). Kształt ten posiada znaczenie w grafice 3D, ponieważ opisuje obszar widoczny w kamerze stosującej rzutowanie perspektywiczne. Stąd bardzo istotna jest możliwość stwierdzenia, czy obiekt (a raczej jego uproszczona bryła otaczająca) przecina to pole widzenia, a tym samym czy wymaga narysowania na ekranie. Moduł dostarcza trzech różnych reprezentacji frustuma, między którymi można przechodzić. Struktura `FRUSTUM_PLANES` opisuje frustum jako 6 płaszczyzn. Struktura `FRUSTUM_POINTS` opisuje go jako 8 punktów. Struktura `FRUSTUM_RADAR` natomiast to tzw. reprezentacja radarowa [61].
- **Sfera** lub kula w 3D nie posiada dedykowanej struktury. Opisywana jest jako punkt — pozycja środka oraz wartość skalarna oznaczająca promień:
`VEC3 SphereCenter; float SphereRadius;`.

Zbiór kilkudziesięciu funkcji oznaczonych jako liczące kolizje pozwala stwierdzać o zachodzeniu na siebie brył różnego rodzaju. Nie sposób opisać dokładnie ich wszystkich, bo temat liczenia kolizji sam w sobie stanowi obszerną dziedzinę. Praktycznie

każda z tych funkcji zawiera w swojej implementacji pewien sprytny algorytm pochodzący z jakiejś książki lub publikacji internetowej, a skompletowanie biblioteki tych funkcji kosztowało wiele czasu i pracy. Wiele z nich zostało opracowanych na podstawie [5]. Dokładne informacje o źródłach poszczególnych algorytmów znaleźć można w komentarzach w kodzie.

Przykładowo, funkcja `SweptBoxToBox` sprawdza kolizję poruszającego się prostopadłościanu z innym prostopadłościanem. Wykorzystuje w tym celu sumę Minkowskiego. Funkcja `TriangleToBox` sprawdza, czy trójkąt w przestrzeni 3D przecina prostopadłościan. Wykorzystuje w tym celu twierdzenie o płaszczyznach rozdzielających (ang. *Separating Axis Theorem*) [62].

Dysk Poissona (ang. *Poisson Disc*, [63]) to takie rozmieszczenie punktów na płaszczyźnie lub w przestrzeni, w którym pozycje tych punktów są losowe, ale żadna para punktów nie jest od siebie odległa o mniej niż określona stała granica. Taki rozkład punktów wykorzystywany bywa w różnych zastosowaniach, np. podczas próbkowania przy śledzeniu promieni (*Supersampling*). Ma tę zaletę ponad regularną siatką punktów, że ich losowe rozmieszczenie zapobiega zjawisku aliasingu, natomiast jego zaleta w porównaniu z rozmieszczeniem zupełnie losowym polega na nieskupianiu się punktów w miejscach o większej lub mniejszej gęstości.

Problem z zastosowaniem dysku Poissona polega na dużej złożoności obliczeniowej generowania zbioru takich punktów. Dlatego dobrze jest przygotować wcześniej tablicę wypełnioną przykładowymi punktami o tym rozkładzie. Problemem jest jednak dostosowanie ich liczby do wymagań konkretnego zastosowania.

Aby temu zaradzić, autor postanowił wygenerować zbiór punktów dysku Poissona wg następującego algorytmu: Losowane są punkty odległe od siebie nie mniej, niż o pewną dużą wartość. Po wielu nieudanych próbach dodania następnego takiego punktu wartość ta jest zmniejszana pozwalając na obecność punktów nieco bliżej siebie położonych. Proces jest powtarzany aż do wygenerowania pożądanej liczby punktów.

Takie podejście pozwala otrzymać tablicę punktów, z których można wziąć $N$ pierwszych elementów i zawsze stanowić one będą poprawny dysk Poissona o wartości granicznej odległości tym większej, im mniejsze jest $N$. Dzięki temu te same tablice punktów zastosowane mogą być w różnych sytuacjach. Autor napisał specjalny program generujący takie tablice za pomocą kosztownych czasowo obliczeń, a następnie zapisał je bezpośrednio w kodzie jako zbiory 100 1–, 2– i 3–wymiarowych punktów w zmiennych o nazwach odpowiednio `POISSON_DISC_1D`, `POISSON_DISC_2D` i `POISSON_DISC_3D`. Proponowana nazwa dla takiego rozwiązania to „progresywny dysk Poissona" (ang. *Progressive Poisson Disc*).

**Moduł Profiler** Profiler w informatyce oznacza narzędzie do analizowania pracy programu w celu zbadania jego wydajności, szczególnie znalezienia „wąskich gardeł", któ-

rych optymalizacji warto poświęcić uwagę. Niniejszym moduł stanowi prostą implementację profilera przeznaczonego do osadzania wprost w kodzie programu. Kod tego modułu zgromadzony jest w plikach `Common\Profiler.hpp` i `Common\Profiler.cpp`.

Najważniejsza klasa — `Profiler` — posiada główną instancję w postaci zmiennej globalnej `g_Profiler`. Można też tworzyć nowe obiekty tej klasy. Wewnętrzne używa ona stosu do monitorowania rejestrowanych w nim zagnieżdżonych operacji i mierzenia czasu ich trwania. Na podstawie takiego pomiaru, najlepiej powtórzonego wielokrotnie, powstaje „profil" — drzewiasta struktura danych opisująca czas trwania poszczególnych operacji i ich podoperacji składowych. Rejestrowania chwili rozpoczęcia i zakończenia operacji dokonuje się wywołując metody odpowiednio `Begin` i `End` lub wygodniej, tworząc obiekt automatyczny za pomocą makra `PROFILE_GUARD`. Po zakończeniu wszystkich tych operacji można otrzymać drzewo z profilem zapisane do pojedynczego łańcucha znaków. Oto przykładowe wyjście generowane przez profiler:

```
Operacja 1 : 14.92 ms (10)
Operacja 2 : 46.8442 ms (10)
  Pod-operacja 1 : 31.2242 ms (10)
  Pod-operacja 2 : 15.4256 ms (10)
```

Oczywiście taki moduł nie zastąpi w pełni prawdziwego zewnętrznego profilera takiego jak AMD CodeAnalyst. Może jednak ułatwić w niektórych sytuacjach badanie wydajności poszczególnych operacji w pisanym kodzie.

**Moduł Stream** Moduł Stream definiuje hierarchię klas strumieni. Kod tego modułu zgromadzony jest w plikach `Common\Stream.hpp` i `Common\Stream.cpp`. W wielu programach odczuwalna jest potrzeba użycia ujednoliconego systemu strumieni, który pozwalałby na zapisywanie i odczytywanie danych bez uwzględniania miejsca, gdzie te dane trafiają (pamięć, plik na dysku itd.). Strumienie tego modułu są przeznaczone do danych binarnych. Ich interfejs jest wzorowany nieco na strumieniach z Delphi, C# i Java. Są za to zupełnie niepodobne do strumieni C++, które stanowią właściwie mieszankę strumieni danych binarnych z prostym parserem potrafiącym przetwarzać tekst na wartości różnego typu. Autor nie jest zwolennikiem takiego podejścia, dlatego do parsowania plików tekstowych przeznaczył osobny moduł — Tokenizer (p. rozdz. 2.3).

Strumienie tego modułu nie są zaprojektowane z myślą o jak najwyższej wydajności. Klasy są polimorficzne i używają metod wirtualnych, a błędy wejścia–wyjścia są zgłaszane poprzez wyjątki modułu Error. Nie ma to jednak aż tak dużego znaczenia, ponieważ przetwarzanie danych przez procesor i tak jest wielokrotnie wolniejsze, niż ich zapis i odczyt z dysku twardego czy nośnika optycznego. Oczywiście zawsze trzeba pamiętać o zasadzie unikania zapisów i odczytów pojedynczych bajtów na rzecz dłuższych bloków pamięci.

Jedną z kluczowych decyzji projektowych było utworzenie osobnych klas bazowych reprezentujących strumienie posiadające i nieposiadające kursora (bieżącej pozycji do

zapisywania/odczytywania danych, którą można zmieniać) i/lub strumieni wejściowych i wyjściowych. Ostatecznie wybrany został ten pierwszy podział. Klasa bazowa wszystkich strumieni — `Stream` — posiada podklasę `SeekableStream` wzbogacającą ją o metody do odczytywania i zapisywania długości danych oraz pozycji kursora. Zdaniem autora jest to wybór oczywisty, ponieważ to czy strumień danego rodzaju posiada kursor (np. plik dyskowy) czy też go nie posiada (np. gniazdo sieciowe) jest rzeczą z góry znaną, podczas gdy możliwość zapisywania i/lub odczytywania danych jest często tylko kwestią odpowiedniej flagi, mówiącej np. o otwieraniu pliku w trybie tylko do odczytu bądź tylko do zapisu.

Poniższy listing prezentuje wykaz metod klasy bazowej `Stream`, dla skrócenia pozbawionych komentarzy objaśniających z pliku nagłówkowego. Pozwalają one na zapisywanie i odczytywanie zarówno dłuższych fragmentów danych binarnych, jak i pojedynczych wartości różnego typu.

```
virtual void Write(const void *Data, size_t Size);
virtual void Flush();
template <typename T> void WriteEx(const T &x)
  { return Write(&x, sizeof(x)); }
void WriteString1(const string &s);
void WriteString2(const string &s);
void WriteString4(const string &s);
void WriteStringF(const string &s);
void WriteBool(bool b);

virtual size_t Read(void *Data, size_t MaxLength);
virtual void MustRead(void *Data, size_t Length);
virtual bool End();
virtual size_t Skip(size_t MaxLength);
template <typename T> void ReadEx(T *x)
  { MustRead(x, sizeof(*x)); }
void ReadString1(string *s);
void ReadString2(string *s);
void ReadString4(string *s);
void ReadStringF(string *s, size_t Length);
void ReadStringToEnd(string *s);
void ReadBool(bool *b);
void MustSkip(size_t Length);

size_t CopyFrom(Stream *s, size_t Size);
void MustCopyFrom(Stream *s, size_t Size);
void CopyFromToEnd(Stream *s);
```

Metody z grupy `WriteStringN` i `ReadStringN` zapisują i odczytują łańcuchy znaków poprzedzone odpowiednio 1, 2 lub 4 bajtami określającymi jego długość. Na uwagę zasługują również szablony metod `WriteEx` i `ReadEx`, które potrafią zapisywać wartości dowolnego typu automatycznie pobierając ich rozmiar.

Poniższy listing prezentuje listę metod, o jakich klasę strumienia wzbogaca podklasa `SeekableStream`.

```
virtual size_t GetSize();
virtual int GetPos();
virtual void SetPos(int pos);
virtual void SetPosFromCurrent(int pos);
virtual void SetPosFromEnd(int pos);
```

```
virtual void Rewind();
virtual void SetSize(size_t Size);
virtual void Truncate();
virtual void Clear();
```

Z klasy bazowej `OverlayStream` wyprowadzone są z kolei tzw. nakładki na strumienie. Nakładka jest strumieniem, który przechowuje wskaźnik do innego strumienia i zapisuje/odczytuje dane do/z niego, a pośrednicząc w ich przesyłaniu przetwarza je przy tym w określony sposób — np. kodując czy kompresując.

Oprócz tych klas podstawowych, moduł definiuje również szereg klas pochodnych pełniących już konkretne funkcje. Oto lista wszystkich klas strumieni z modułu Stream:

- `Stream` — klasa bazowa strumieni.
- `SeekableStream` — klasa bazowa strumieni z obsługą długości i kursora.
- `CharWriter` — klasa przyspieszająca zapisywanie do strumienia po znaku.
- `CharReader` — klasa przyspieszająca odczytywanie ze strumienia po znaku.
- `MemoryStream` — strumień do bloku pamięci o stałym rozmiarze.
- `VectorStream` — strumień do samorozszerzającego się bloku pamięci.
- `StringStream` — strumień do łańcucha typu `string`.
- `OverlayStream` — klasa bazowa nakładek na strumienie.
- `CounterOverlayStream` — nakładka zliczająca zapisywane i odczytywane dane.
- `LimitOverlayStream` — nakładka ograniczająca ilość zapisywanych i odczytywanych danych.
- `MultiWriterStream` — strumień zapisujący na raz do wielu strumieni.
- `Hash_Calc` — strumień liczący hash.
- `CRC32_Calc` — strumień liczący sumę kontrolną CRC32.
- `MD5_Calc` — strumień liczący sumę kontrolną MD5.
- `XorCoder` — strumień szyfrujący i deszyfrujący dane operacją XOR.
- `BinEncoder`, `BinDecoder` — strumień kodujący/dekodujący dane binarne jako ciąg zer i jedynek. Każdy bajt zamienia na 8 znaków.
- `HexEncoder`, `HexDecoder` — strumień kodujący/dekodujący dane binarne jako ciąg liczb szesnastkowych. Każdy bajt zamienia na 2 znaki.
- `Base64Encoder`, `Base64Decoder` — strumień kodujący/dekodujący dane binarne w formacie Base64. Każde 3 bajty zamienia na 4 znaki.

Moduł Stream definiuje też strukturę `MD5_SUM` reprezentującą sumę kontrolną MD5, a także jej konwersję do i z łańcucha. Inne moduły — Files i ZlibUtils — rozszerzają hierarchię strumieni o nowe klasy.

**Moduł Threads** Moduł Threads dostarcza przenośnych klas do wielowątkowości i synchronizacji. Kod tego modułu zgromadzony jest w plikach `Common\Threads.hpp` i `Common\Threads.cpp`.

Utworzenie biblioteki wspierającej programowanie równoległe, która byłaby przenośna na Windows i Linux, stanowi pewien problem. W każdym z tych systemów są bowiem używane inne wywołania (w Linuksie używana jest biblioteka pthreads, a Windows posiada własne funkcje systemowe). Te dwa API różnią się nie tylko nazwami funkcji, ale posiadają zupełnie inne prymitywy synchronizujące. Autorowi udało się jednak zaimplementować wszystkie z nich na obydwu platformach, emulując je w razie braku natywnego wsparcia ze strony systemu za pomocą innych. Mechanizmy systemowe użyte do realizacji poszczególnych prymitywów pokazuje tabela 2.3. Podczas implementacji bardzo pomocna była książka [64]. Oto opis dostępnych klas:

Tablica 2.3. Implementacja prymitywów synchronizujących modułu Threads na poszczególnych platformach.

| | Windows | Linux |
|---|---|---|
| Mutex | `CRITICAL_SECTION`, Mutex | `pthread_mutex_t` |
| Semaphore | Semaphore | `sem_t` |
| Cond | (emulowany) | `pthread_cond_t` |
| Barrier | (emulowany) | `pthread_barrier_t` |
| Event | Event | (emulowany) |

- `Thread` to klasa bazowa wątku. Aby utworzyć swój wątek, należy po niej odziedziczyć, podobnie jak w języku Java.
- `Mutex` (od ang. *Mutually Exclusive*) to obiekt synchronizujący zapewniający, że objętą nim sekcję krytyczną kodu wykonuje w danej chwili co najwyżej jeden wątek. Pozostałe, które próbują to zrobić, muszą poczekać.
- `Semaphore` to semafor — obiekt synchronizujący, który posiada wewnętrzny licznik. Operacja `V` to podniesienie semafora, zwiększa licznik o 1. Operacja `P` to opuszczenie semafora, zmniejsza licznik o 1, a jeśli jego wartość wynosi 0, czeka aż inny wątek go podniesie. Semafor może zatem służyć do ograniczania liczby wątków mających jednoczesny dostęp do pewnego zasobu.
- `Cond` to zmienna warunkowa (ang. *Conditional Variable*). Wątek może wywołać jej metodę `Wait` czekając tym samym, aż inny wątek wznowi jego działanie wywołując metodę `Signal` lub `Broadcast`. Zmiennej warunkowej używa się często w określony sposób w połączeniu z pewnym własnym warunkiem (stąd nazwa). Może służyć do realizacji wzorca monitora. Implementacja tego prymitywu pod Windows została opracowana na podstawie kodu biblioteki wxWidgets i z pomocą [65].
- `Barrier` to bariera — obiekt synchronizujący, który może stanowić punkt synchronizacji dla określonej liczby równolegle wykonywanych wątków. Wywołanie metody `Wait` blokuje wątek do czasu, aż określona liczba wątków zostanie w ten sposób zablokowana. Dopiero wtedy wszystkie na raz zostają odblokowane.
- `Event` (zdarzenie) to obiekt posiadający stan w postaci wartości logicznej. Może być w stanie ustawionym lub nieustawionym. Pośród jego metod znajduje się

czekanie aż zdarzenie przejdzie w stan ustawiony, jego przestawianie do stanu ustawionego lub nieustawionego. Zdarzenie może zostać utworzone w trybie, w którym samo przestawia się z powrotem na stan nieustawiony po odblokowaniu czekającego na nim wątku.

**Moduł Tokenizer** W profesjonalnej produkcji gier i silników, w której z tworzonego systemu informatycznego korzysta wiele osób i nie wszyscy z nich są programistami, ważne jest przygotowanie łatwych w użyciu narzędzi i edytorów wyposażonych w graficzny interfejs użytkownika. W warunkach amatorskich jednak wystarczające jest, a przy tym dużo prostsze i szybsze do zrealizowania, zapisywanie pewnych informacji w specjalnie sformatowanych plikach tekstowych odczytywanych bezpośrednio przez program. Zachodzi więc potrzeba definiowania wielu takich formatów plików i ich łatwego parsowania w kodzie silnika.

Do generowania parserów wykorzystywać można specjalne narzędzia takie jak lex i yacc. Interpretowanie plików tekstowych, czy to języków opisu czy też języków programowania, składa się zwykle z dwóch etapów — tokenizacji (rozbicia łańcucha znaków na tokeny) i parsowania (analizy składniowej). Zdaniem autora najwygodniejszym podejściem do tego zagadnienia jest dostarczenie gotowego tokenizera o interfejsie tak prostym, aby tworzenie parserów konkretnego języka z jego użyciem było wygodne. Ponadto, języki opisu przeznaczony do różnych formatów plików (np. do opisywania materiałów, konfiguracji programu, listy zasobów itd.) łatwiej jest opanować, jeśli wszystkie oparte są na podobnej składni. Dlatego powstał moduł Tokenizer.

Jego kod znajduje się w plikach `Common\Tokenizer.hpp` i `Common\Tokenizer.cpp`. Zapewniana przez niego klasa `Tokenizer` przyjmuje w konstruktorze łańcuch lub strumień ze znakami do parsowania. Wywołanie metody `Next` odczytuje następny token i zapamiętuje go w prywatnych polach klasy. Jego typ i wartość można stamtąd pobrać metodami takimi jak `GetToken`, `GetString`, `GetUint4`, `GetFloat` itd. Klasa zapewnia wiele dodatkowych metod ułatwiających parsowanie. Przykładowo, wywołanie metody `AssertIdentifier` sprawdza, czy ostatnio odczytany token jest identyfikatorem o podanej treści. Jeśli nie jest, metoda rzuca wyjątek sygnalizujący błąd, automatycznie wypełniony odpowiednim komunikatem i wskazujący mejsce wystąpienia błędu (numer znaku, wiersza i kolumny).

Opisywany moduł rozbija tekst na tokeny wg składni podobnej do języka C i C++. Obsługiwane rodzaje tokenów to: symbole (np. `+`, `$`, `.`), identyfikatory (np. `Abc`), liczby całkowite i zmiennoprzecinkowe (np. `123`, `0xFF`, `-3.14`), stałe znakowe i łańcuchowe (np. `'A'` `"ABC"`). Sekwencje ucieczki i komentarze wyglądają tak samo, jak w języku C i C++.

Tworzenie parsera zaprojektowanego wcześniej formatu tekstowego (języka opisu) z użyciem tego tokenizera jest bardzo proste. Na przykład jeśli plik wygląda tak:

```
{
  Nazwa1 = "Wartość1"
  Nazwa2 = "Wartość2"
}
```

Wówczas kompletna funkcja rozkładająca tekst na dane może wyglądać następująco:

```
void ParseMyFormat(std::map<string, string> &Out, const string &In)
{
  Tokenizer t(&In, 0);
  t.Next();
  t.AssertSymbol('{');
  t.Next();
  std::pair<string, string> Item;
  while (!t.QuerySymbol('}'))
  {
    t.AssertToken(Tokenizer::TOKEN_IDENTIFIER);
    Item.first = t.GetString();
    t.Next();
    t.AssertSymbol('=');
    t.Next();
    t.AssertToken(Tokenizer::TOKEN_STRING);
    Item.second = t.GetString();
    t.Next();
    Out.insert(Item);
  }
  t.Next();
  t.AssertEOF();
}
```

**Moduł ZlibUtils** Ten dodatkowy, opcjonalny moduł jest nakładką na bibliotekę zlib. Jego kod znajduje się w plikach `Common\ZlibUtils.hpp` i `Common\ZlibUtils.cpp`. Zlib to biblioteka do kompresji danych algorytmem Deflate, używanym m.in. w formatach plików PNG i GZ. Jej interfejs, mający postać zbioru kilku funkcji języka C, jest bardzo trudny do opanowania i poprawnego użycia. Dlatego zachodzi potrzeba obudowania go w bardziej wygodne, obiektowe nakładki.

Moduł definiuje następujące klasy:

- `ZlibError` to klasa wyjątku reprezentująca błąd zgłaszany przez bibliotekę zlib.
- `ZlibCompressionStream` to strumień kompresji danych do formatu zlib.
- `GzipCompressionStream` to strumień kompresji danych do formatu gzip.
- `ZlibDecompressionStream` to strumień dekompresji danych z formatu zlib.
- `GzipDecompressionStream` strumień dekompresji danych z formatu gzip.
- `GzipFileStream` to strumień zapisu i odczytu pliku w formacie gzip (zalecane rozszerzenie: GZ).

## 2.4. Architektura szkieletu

Szkielet (ang. *Framework*) to warstwa dostarczająca podstawowej funkcjonalności potrzebnej w każdym programie korzystającym z Direct3D — w tym tworzenia okna Windows, inicjalizacji Direct3D, obsługi wejścia z klawiatury i myszki, zarządzania zasobami. Składa się na niego kod zgromadzony w katalogu `Framework`. Listę modułów tej warstwy pokazuje tabela 2.4.

Tablica 2.4. Alfabetyczna lista modułów szkieletu.

| | |
|---|---|
| AsyncConsole | Asynchroniczna obsługa konsoli systemowej. |
| D3dUtils | Elementy wspomagające pracę z Direct3D. |
| Framework | Główny moduł — szkielet aplikacji. |
| Gfx2D | Moduł do rysowania grafiki 2D. |
| GUI | Podstawa systemu GUI. |
| GUI_Controls | Kontrolki systemu GUI. |
| GUI_PropertyGridWindow | Kontrolka PropertyGrid. |
| Multishader | Zasób shadera komplowanego warunkowo. |
| QMesh | Zasób siatki 3D z obsługą animacji szkieletowej. |
| Res_d3d | Klasy zasobów związanych z Direct3D. |
| ResMngr | Podstawowe klasy managera zasobów. |

Cały kod szkieletu i warstw wyższych (nazwanych Silnik i Gra) korzysta z mechanizmu nagłówka prekompilowanego (ang. *Precompiled Header*), który znacznie przyspiesza kompilację. Przeznaczone dla niego pliki to `Framework\pch.hpp` i `Framework\pch.cpp`.

Szkielet korzysta z biblioteki bazowej, natomiast nie wymaga do działania warstw wyższych, w tym silnika. Dzięki temu może zostać wykorzystany w różnorodnych zastosowaniach — np. do wizualizacji naukowych, gier 2D i 3D. Nie jest przenośny — działa tylko w środowisku Windows.

**Moduł AsyncConsole** Wiele silników (np. te użyte w grach Quake i Neverwinter Nights) posiada konsolę tekstową pozwalającą na wypisywanie komunikatów i wprowadzanie poleceń. Często jest to konsola rysowana we własnym zakresie ponad obrazem renderowanym przez silnik. Autor zdecydował się jednak na prostsze rozwiązanie — użycie dodatkowego okna standardowej konsoli systemowej.

Funkcje do obsługi konsoli systemowej z biblioteki standardowej C (jak `printf`, `scanf`) czy C++ (jak `std::cout`, `std::cin`) mają liczne ograniczenia. Nie wspierają obsługi kolorów, jak również nie pozwalają na pracę asynchroniczną (oczekiwanie na wprowadzenie polecenia blokuje program).

Dlatego moduł AsyncConsole, którego kod zgromadzony jest w plikach `Framework\AsyncConsole.hpp` i `Framework\AsyncConsole.cpp`, do obsługi konsoli systemowej korzysta z funkcji WinAPI. Zdefiniowana w nim klasa `AsyncConsole` daje przez prosty interfejs dostęp do funkcji takich jak wypisywanie komunikatów wraz z obsługą kolorów, jak również sprawdzanie polecenia wpisanego przez użytkownika. Odbieranie poleceń jest asynchroniczne, tzn. użytkownik może wprowadzać tekst do konsoli w dowolnej chwili, a program może sprawdzać wprowadzone polecenia bez blokowania swojego wykonywania. Zostało to osiągnięte poprzez użycie osobnego wątku czekającego na polecenia. Klasa jest przy tym bezpieczna wątkowo umożliwiając równoczesny dostęp do konsoli różnym częściom programu.

**Moduł D3dUtils** Kod zgromadzony w plikach `Framework\D3dUtils.hpp` i `Framework\D3dUtils.cpp` to zbiór struktur i funkcji wspomagających pracę z biblioteką Direct3D.

Kolekcja 16 struktur (takich jak np. `VERTEX_XN2`) stanowi definicję najczęściej używanych formatów wierzchołka. Każda z nich posiada pola odpowiadające swojej nazwie (np. tu `X` oznacza pozycję 3D, `N` wektor normalny, a `2` dwuwymiarowe współrzędne tekstury), jak również definiuje flagę bitową FVF (ang. *Flexible Vertex Format*) opisującą dany format wierzchołka.

Szereg klas stanowi otoczkę na funkcje Direct3D zgodnie ze wzorcem RAII, czyli dokonując automatycznej finalizacji w swoim destruktorze. Ich użycie zapewnia mniejsze ryzyko popełnienia przez programistę błędu polegającego na pominięciu finalizacji. Klasa `RenderTargetHelper` ustawia w urządzeniu Direct3D podany cel renderowania (ang. *Render Target*) i/lub bufor głębokości i szablonu (ang. *Depth–Stencil Buffer*), a w destruktorze przywraca oryginalne ustawienia. Klasa `D3dxBufferWrapper` przechowuje wskaźnik do interfejsu `ID3DXBuffer` (używanego w Direct3D m.in. do zgłaszania komunikatów o błędach kompilacji shaderów) i automatycznie zwalnia go w destruktorze. Klasy `VertexBufferLock` i `IndexBufferLock` służą do blokowania buforów wierzchołków i indeksów, automatycznie odblokowując je w destruktorze.

Kilka funkcji przeznaczonych jest do operowania na formacie wierzchołka FVF. Przykładowo `CalcComponentSizesAndOffsets` potrafi obliczyć rozmiary i offsety poszczególnych elementów wierzchołka (jak pozycja, wektor normalny, koordynaty tekstury) na podstawie podanego formatu FVF.

Moduł posiada także funkcje służące do konwertowania różnych typów wyliczeniowych używanych przez Direct3D do/z łańcuchów znaków. Mogą one stanowić pomoc przy implementowaniu wczytywania i zapisywania konfiguracji graficznej programu. Przykładowo, funkcja `StrToD3dfmt` zamieni łańcuchy o treści `"A8R8G8B8"` i `"D3DFMT_A8R8G8B8"` na wartość stałej `D3DFMT_A8R8G8B8`.

**Moduł Framework** Ten moduł, którego kod zgromadzony jest w plikach `Framework\Framework.hpp` i `Framework\Framework.cpp`, stanowi jakby podstawę całego programu. To do niego kierowane jest sterowanie z funkcji głównej `WinMain`, a w jego wnętrzu znajduje się pętla główna programu z obsługą komunikatów okna Windows. Wszystkie zdefiniowane w nim elementy zgromadzone są w przestrzeni nazw `frame`. Funkcje tego modułu to:

- tworzenie, usuwanie, zarządzanie oknem Windows,
- obsługa okna — sterowanie jego możliwościami, odbieranie komunikatów (skalowanie, minimalizacja, utrata aktywności itp.),
- tworzenie, usuwanie, resetowanie obiektu Direct3D oraz urządzenia Direct3D,
- obsługa utraty urządzenia D3D,
- udostępnianie do użytku: uchwytu okna, obiektu D3D, urządzenia D3D,

- zmiana ustawień graficznych w czasie pracy,
- przechwytywanie i obsługa błędów modułu Error,
- obsługa pęli komunikatów,
- obsługa wejścia z klawiatury i z myszy,
- pomiar czasu i zliczanie FPS,
- wczytywanie z pliku i zapisywanie do pliku ustawień graficznych.

Poniższy listing prezentuje pola struktury opisującej ustawienia graficzne.

```
struct SETTINGS
{
  uint4 BackBufferWidth;
  uint4 BackBufferHeight;
  uint4 RefreshRate;
  D3DFORMAT BackBufferFormat;
  uint4 BackBufferCount;
  D3DMULTISAMPLE_TYPE MultiSampleType;
  uint4 MultiSampleQuality;
  D3DSWAPEFFECT SwapEffect;
  bool FullScreen;
  bool EnableAutoDepthStencil;
  D3DFORMAT AutoDepthStencilFormat;
  bool DiscardDepthStencil;
  bool LockableBackBuffer;
  uint4 PresentationInverval;
  enum FLUSH_MODE { FLUSH_NONE, FLUSH_EVENT, FLUSH_LOCK };
  FLUSH_MODE FlushMode;
};
```

Większość z tych pól odpowiada bezpośrednio polom struktury `D3DPRESENT_PARAMETERS`, opisującej ustawienia graficzne Direct3D, więc nie wymaga wyjaśnienia. Na uwagę zasługuje pole wyliczeniowe typu `FLUSH_MODE`. Oznacza ono sposób, w jaki program ma wymusić poczekanie na zakończenie renderowania przez kartę graficzną. Domyślnie takie czekanie nie powinno być wykonywane, gdyż spowalnia tylko działanie całego programu. Autor napotkał jednak błąd w sterownikach graficznych firmy NVIDIA, który w pewnych warunkach powodował, że karta nie dokańczała renderowania całej wysłanej do niej geometrii przed zaprezentowaniem klatki na ekranie, co owocowało znikaniem bądź migotaniem niektórych obiektów albo ich fragmentów.

Dlatego szkielet posiada możliwość włączenia wymuszonego czekania na skończenie pracy karty graficznej przed zaprezentowaniem nowej klatki na ekranie. Wartość `FLUSH_EVENT` powoduje użycie w tym celu obiektu zapytania `IDirect3DQuery9` typu `D3DQUERYTYPE_EVENT`. Ponieważ w pewnych publikacjach pojawiają się głosy, że nawet ta metoda nie zawsze jest skuteczna, wartość `FLUSH_LOCK` pozwala zastosować bardziej „brutalną” metodę korzystającą z blokowania specjalnie w tym celu stworzonych powierzchni (ang. *Surface*) [66].

Sposób działania programów komputerowych można ogólnie podzielić na 3 modele.

1. Pierwszy, który można nazwać „liniowym”, polega na odczytaniu danych wejściowych, dokonaniu pewnych obliczeń, zapisaniu danych wyjściowych i zakoń-

czeniu pracy. Stosowany bywa np. w obliczeniowych programach naukowych, w prostych programach dydaktycznych czy w skryptach administracyjnych.
2. Drugi model, który można nazwać „sterowanym zdarzeniami", polega na oczekiwaniu w pętli na polecenia użytkownika i wykonywaniu tych poleceń. Jest charakterystyczny dla programów z interfejsem okienkowym.
3. Trzeci model jest wykorzystywany w grach. Można go nazwać modelem „czasu rzeczywistego". Polega na wykonywaniu w pętli nieustannych obliczeń polegających na aktualizacji stanu symulacji na podstawie kroku czasowego, a także odrysowaniu nowej klatki obrazu na ekranie. W każdej iteracji pętli jest też sprawdzane wejście od użytkownika, ale program nie oczekuje na komunikaty wejściowe.

Prezentowany tu szkielet aplikacji może działać w trybie 2 lub 3. Zmiana modelu następuje przez wywołanie funkcji `SetLoopMode`. Podanie jako parametru `false` powoduje, że program nie będzie wykonywał nieustannie pętli, tylko oczekiwał na wejście od użytkownika, a co za tym idzie nie będzie wykorzystywał całej dostępnej mocy obliczeniowej procesora. To dobre rozwiązanie dla programów okienkowych, edytorów czy gier turowych. Z drugiej strony, gry i symulacje czasu rzeczywistego bądź jakiekolwiek prezentujące animacje muszą działać w domyślnym trybie, w którym pętla nie blokuje się czekając na komunikaty od użytkownika.

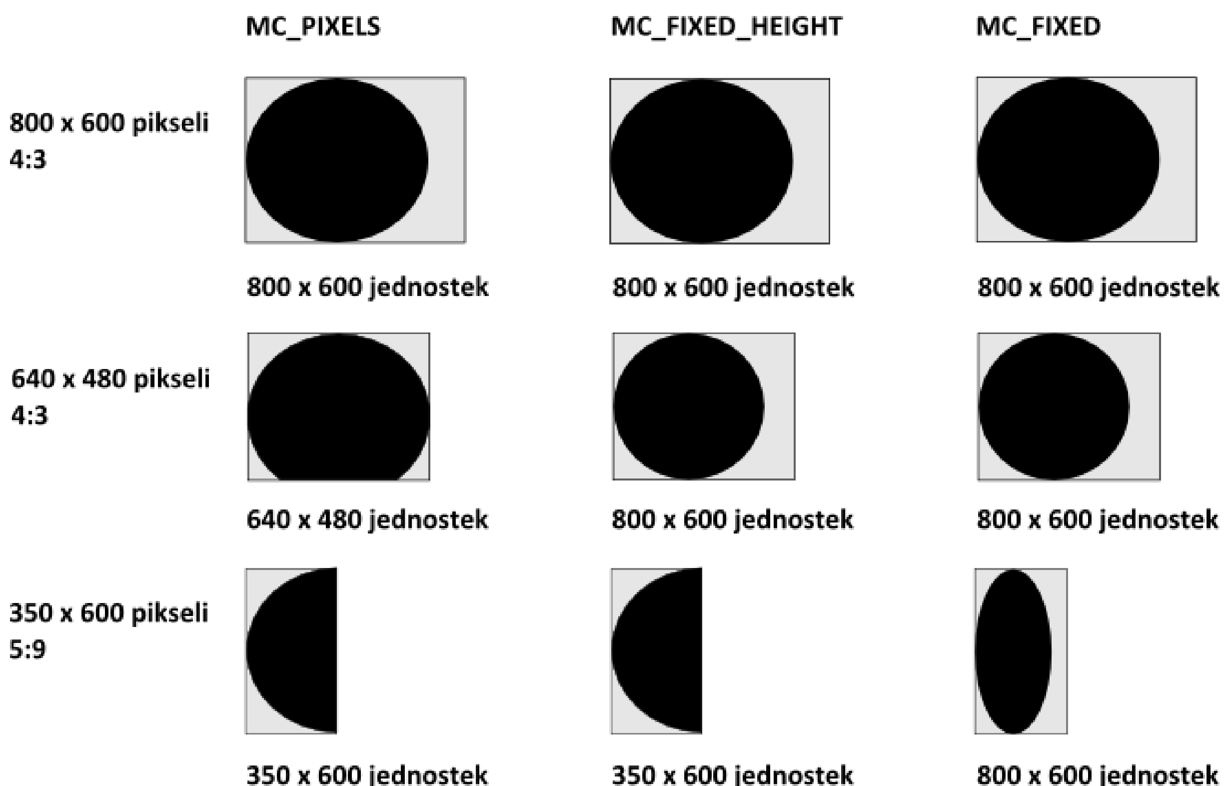
Aby utworzyć obiekt odbierający wywołania zwrotne dotyczące zdarzeń inicjalizacji i finalizacji całego programu, jak również utraty/odzyskania urządzenia, minimalizacji/przywrócenia i deaktywacji/aktywacji jego okna, należy odziedziczyć po klasie `IFrameObject` i zarejestrować obiekt funkcją `RegisterFrameObject`. Listę metod tego interfejsu pokazuje poniższy listing:

```
class IFrameObject
{
public:
  virtual void OnCreate();
  virtual void OnDestroy();
  virtual void OnLostDevice();
  virtual void OnResetDevice();
  virtual void OnRestore();
  virtual void OnMinimize();
  virtual void OnActivate();
  virtual void OnDeactivate();
  virtual void OnResume();
  virtual void OnPause();
  virtual void OnTimer(uint4 TimerId);
};
```

Analogicznie wygląda odbieranie wywołań zwrotnych z komunikatami od klawiatury i myszki. Aby to zrobić, należy odziedziczyć po klasie `IInputObject` i zarejestrować obiekt funkcją `RegisterInputObject`. Wspomniany interfejs posiada następujące metody:

```
class IInputObject
{
public:
  virtual bool OnKeyDown(uint4 Key);
  virtual bool OnKeyUp(uint4 Key);
  virtual bool OnChar(char Ch);
  virtual bool OnMouseMove(const VEC2 &Pos);
  virtual bool OnMouseButton(const VEC2 &Pos, MOUSE_BUTTON Button,
    MOUSE_ACTION Action);
  virtual bool OnMouseWheel(const VEC2 &Pos, float Delta);
};
```

Omawiany moduł dostarcza również układu współrzędnych do rysowania wszelkiej grafiki 2D. W tym układzie podaje też pozycję kursora myszy. Na definicję tego układu współrzędnych składa się określona szerokość, wysokość i wybrany jeden spośród 3 możliwych trybów. W zależności od niego inne jest mapowanie pikseli na wirtualne jednostki, tak jak to pokazuje rys. 2.3.




Rys. 2.3. Rozdzielczość ekranu 2D w wirtualnych jednostkach dla podanych parametrów `MouseCoordsWidth`= 800, `MouseCoordsHeight`= 600 i różnych trybów pracy.

Ponadto, mysz może pracować w „trybie kamery”. Można ten tryb uaktywnić funkcją `SetMouseMode`. Wówczas zwracana jest nie pozycja kursora, ale jego względne przesunięcie od położenia poprzedniego. Zostało to zrealizowane poprzez przemieszczanie kursora po każdym ruchu z powrotem na środek ekranu. Dzięki temu myszkę w tym trybie można przesuwać w każdą stronę o dowolną odległość bez ograniczenia krawędziami ekranu. Taki tryb ma zastosowanie do sterowania kamerą, np. w grach FPP (z perspektywy pierwszej osoby).

Szkielet dokonuje także pomiaru czasu. Korzysta w tym celu z odpowiednich możliwości biblioteki bazowej. Warstwom wyższym natomiast udostępnia obiekty `Timer1` i `Timer2`, z których można pobierać czas, jaki minął od uruchomienia programu oraz krok czasowy od poprzedniej klatki, wyrażone jako liczby typu `float`, w sekundach. Każdy z tym zegarów można zatrzymywać i wznawiać, posiadają one wewnętrzne liczniki zatrzymań. `Timer2` jest przeznaczony do sterowania właściwą symulacją, podczas gdy z obiektu `Timer1` należy korzystać przy animowaniu elementów interfejsu użytkownika itp. Dzięki temu rozróżnieniu każdym z tych zegarów można sterować osobno, np. zatrzymać symulację, ale nie cały interfejs programu. Kod pomiaru czasu dodatkowo wylicza średnią liczbę klatek na sekundę (FPS — ang. *Frames Per Second*), która jest najczęściej używaną miarą wydajności w grach. Można tą wartość pobrać za pomocą funkcji `GetFPS`.

Szkielet zajmuje się też wyłapywaniem i obsługą błędów w postaci wyjątków. Musi to robić w wielu miejscach swojego kodu, ponieważ wyjątek C++ nie może się dostać do kodu WinAPI. Każdy taki błąd jest zapisywany do pliku `Errors.log` i powoduje zakończenie działania programu (jednak nie jego przerwanie, ale normalne zamknięcie z wywołaniem zwalniania wszelkich zaalokowanych obiektów).

**Manager zasobów** **Zasób** można zdefiniować jako każdy taki obiekt, którego utworzenie wraz z zawartymi w nim danymi wymaga czaso– i pracochłonnej operacji, najczęściej wczytania pliku z dysku, ewentualnie wygenerowania proceduralnego. Przykładami zasobów w kontekście grafiki 3D są tekstury, siatki trójkątów czy shadery. Manager zasobów, jako moduł przechowujący kolekcję zasobów i dający do nich dostęp, jest ważną częścią każdego szkieletu i silnika graficznego. Jego implementacja wiąże się z podjęciem kilku decyzji projektowych.

Jedną z takich decyzji jest napisanie osobnych managerów przeznaczonych dla poszczególnych rodzajów zasobów — np. tekstur, siatek trójkątów, shaderów itp. bądź jeden uogólniony manager wszystkich zasobów. Autor zdecydował się na to drugie rozwiązanie. Jego kod znajduje się w plikach `Framework\ResMngr.hpp` i `Framework\ResMngr.cpp`. Obiekt managera jest ogólnodostępny pod globalnym wskaźnikiem `g_Manager`. Wszelkie typy zasobów są klasami dziedziczącymi z `IResource`.

Drugą decyzją projektową jest sposób dostępu do zasobów. Istnieje wiele rozwiązań tego problemu, a każde ma swoje zalety i wady. Najbardziej intuicyjne jest skojarzenie z każdym zasobem nazwy jako łańcucha znaków. Wyszukiwanie zasobów po nazwie jest jednak kosztowne obliczeniowo. Dlatego ulepszenie tej metody może polegać na zapamiętywaniu jedynie hasha z nazwy. Inny sposób to nadawanie zasobom identyfikatorów liczbowych lub uchwytów. Autor niniejszej pracy zdecydował się na następujące rozwiązanie: Zasoby są identyfikowane przez nazwy i pamiętane w strukturze danych `std::map<string, IResource*>`, która pozwala na ich względnie szybkie

wyszukiwanie. Ponadto każdy zasób istnieje jako obiekt zawsze, niezależnie od tego czy jego dane są wczytane. Dzięki temu wyszukiwanie zasobu po nazwie można przeprowadzać tylko raz (podczas tworzenia obiektu który korzysta z zasobu lub podczas pierwszego użycia zasobu), a potem już posługiwać się wprost wskaźnikiem do zasobu.

Ponieważ zasoby mogą być różnych typów i są przechowywane razem, zachodzi potrzeba rzutowania w dół obiektów klasy bazowej `IResource` na klasy pochodne. Aby uczynić wyszukiwanie zasobu prostszym i wygodniejszym, rzutowanie to wraz ze sprawdzaniem typu za pomocą RTTI jest ukryte w szablonie metody:

```
template <typename T>
T * MustGetResourceEx(const string &Name)
{
  IResource *R = MustGetResource(Name);
  if (typeid(*R) != typeid(T)) throw Error(...);
  return static_cast<T*>(R);
}
```

Istnieje wiele sposobów na używanie zasobów. Najprostszy polega na wczytaniu listy zasobów i danych każdego z tych zasobów podczas uruchamiania programu. W programach z dużą liczbą zasobów, takich jak współczesne gry, jest to jednak niemożliwe i zasoby trzeba ładować oraz zwalniać w razie potrzeby — podczas specjalnej fazy ładowania lub już podczas normalnej pracy programu, np. przy pierwszym użyciu. Opisywany manager zasobów pozwala na takie operacje. Dalesze rozszerzenie tych możliwości polegałoby na wczytywaniu zasobów w tle. Jest to potrzebne w pewnego typu grach, jednak znacznie komplikuje budowę managera zasobów (konieczne jest użycie osobnego wątku i odpowiednia synchronizacja), dlatego opisywany tutaj manager nie posiada takiej możliwości.

Lista zasobów może się znajdować w specjalnym pliku tekstowym. Dzięki temu można ją zmieniać bez rekompilacji programu. Format takiego pliku został zbudowany w oparciu o tokenizer (patrz p. 2.3) i jest rozszerzalny w taki sposób, że nowe typy zasobów mają obowiązek dostarczać metody statycznej parsującej parametry zasobu tego typu. Ponadto istnieje możliwość dynamicznego tworzenia i usuwania zasobów w czasie pracy programu z poziomu jego kodu, bez użycia opisu tekstowego.

Każdy zasób (obiekt klasy pochodnej od `IResource`) ma swój stan oznaczający, czy jego dane są wczytane. Jeśli zasób jest wczytany (można to sprawdzić metodą `IsLoaded`), użytkownikowi wolno pobierać te dane metodami charakterystycznymi dla zasobu danego rodzaju. Wczytania zasobu można dokonać metodą `Load`, a zwolnienia jego danych metodą `Unload`. Zasób nieużywany przez określony czas (domyślnie 2 minuty) jest automatycznie zwalniany. Istnieje możliwość zablokowania zasobu metodą `Lock`, co daje gwarancję, że pozostanie on wczytany aż do odblokowania metodą `Unlock`. Zasób posiada wewnętrzny licznik zablokowań.

Te funkcje można wykorzystać na jeden z dwóch typowych sposobów:

1. Jeśli zasobów jest niewiele i program potrzebuje ich wszystkich do działania, może zablokować każdy zasób metodą `Lock` lub zdefiniować zasoby jako stale

zablokowane w ich definicji w pliku tekstowym. Zasoby są ładowane w chwili tego wywołania i pozostają załadowane aż do zakończenia programu.
2. Jeśli zasobów jest dużo, program może wczytywać tylko te potrzebne w danej chwili, przed ich użyciem. Musi w tym celu wywoływać metodę `Load` zasobu przed każdym jego użyciem, gdyż pozostanie zasobu w stanie wczytanym po tym wywołaniu jest gwarantowane tylko do końca danej klatki. Potem manager może zdecydować o zwolnieniu zasobu.

Zadaniem klasy pochodnej reprezentującej konkretny typ zasobu jest dostarczenie implementacji metod abstrakcyjnych: `OnLoad` służącej do wczytania zasobu, `OnUnload` służącej do zwolnienia jego danych i opcjonalnie `OnEvent`, oznaczającej dodatkowe zdarzenia. To dodatkowe zdarzenie jest wykorzystane do powiadomienia zasobów o utracie i odzyskaniu urządzenia Direct3D, co zmusza do ponownego wczytania zasobów Direct3D umieszczonych w puli pamięci `DEFAULT`. Inne rodzaje zasobów mogą to zdarzenie zignorować.

Pliki `Framework\Res_d3d.hpp` i `Framework\Res_d3d.cpp` dostarczają implementacji klas zasobów związanych z Direct3D. Klasą bazową wszystkich takich zasobów jest `D3dResource`. Tłumaczy ona wywołania związane z zasobami ogólnie — `OnLoad`, `OnUnload` i `OnEvent` — na wywołania charakterystyczne dla zasobów Direct3D, których zaimplementowanie pozostawia klasom pochodnym: `OnDeviceCreate`, `OnDeviceDestroy`, `OnDeviceRestore`, `OnDeviceInvalidate`. Klasy pochodne dostarczane przez wspomniane wyżej pliki to:

- `D3dBaseTexture` to klasa bazowa dla wszelkiego rodzaju tekstur Direct3D.
- `D3dTexture` reprezentuje zwyczajną, dwuwymiarową teksturę.
- `D3dCubeTexture` reprezentuje teksturę sześcienną.
- `D3dFont` reprezentuje czcionkę `ID3DXFont`.
- `D3dEffect` reprezentuje efekt `ID3DXEffect`.
- `Font` reprezentuje własną implementację czcionki wczytywanej na podstawie pliku graficznego z obrazem znaków i dodatkowego pliku tekstowego z opisem, tworzonych przez osobne narzędzie — Bitmap Font Generator.
- `D3dTextureSurface` reprezentuje teksturę i/lub powierzchnię D3D używaną jako cel renderowania.
- `D3dCubeTextureSurface` reprezentuje teksturę i/lub powierzchnię sześcienną D3D używaną jako cel renderowania.
- `D3dVertexBuffer` reprezentuje bufor wierzchołków D3D.
- `D3dIndexBuffer` reprezentuje bufor indeksów D3D.
- `OcclusionQueries` reprezentuje kolekcję zapytań zapytań o zasłanianie (ang. *Occlusion Query*) i wirtualizuje ich pulę udostępniając dowolną liczbę „wirtualnych” zapytań.

Inne moduły rozszerzają zbiór klas zasobów o dodatkowe, bardziej rozbudowane typy. Opisane są one w następnych podrozdziałach.

**Moduł QMesh** Moduł, którego kod znajduje się w plikach `Framework\QMesh.hpp` i `Framework\QMesh.cpp`, definiuje klasę `QMesh`. Jest ona rodzajem zasobu, który reprezentuje model, czyli siatkę trójkątów wraz z dodatkowymi informacjami stanowiącą reprezentację kształtu jakiegoś obiektu. Jest to właściwie najważniejszy typ zasobu — grafika 3D prawie w całości składa się z takich siatek trójkątów [3].

Klasa potrafi wczytywać plik w formacie QMSH, zaprojektowanym specjalnie na potrzeby niniejszej pracy (patrz rozdz. 3.3). Pliki tego formatu przygotowywane są na podstawie formatu pośredniego QMSH.TMP przez narzędzie konsolowe Tools (patrz rozdz. 2.6). Wewnętrznie do przechowywania wczytanych informacji klasa używa bufora wierzchołków i bufora indeksów Direct3D oraz dodatkowych struktur przechowujących odpowiednie metadane.

Oprócz udostępniania na zewnątrz wczytanych informacji oraz dostępu do buforów, które potrzebne są podczas renderowania siatki, klasa wspiera także *Skinning*, czyli animację szkieletową z wagami (patrz 3.4). Model może posiadać zapisany szkielet, który składa się z hierarchii kości, a każdy wierzchołek ma wówczas przypisane numery co najwyżej dwóch kości które mają na niego wpływ wraz z ich wagami.

W celu zrealizowania animacji szkieletowej klasa udostępnia metodę `GetBoneMatrices` zwracającą tablicę macierzy poszczególnych kości w określonej pozycji. Macierze te można przekazać jako stałe do Vertex Shadera, aby wykonać skinning sprzętowo na GPU. Klasa potrafi jednak liczyć skinning również na CPU i wykonuje go wewnętrznie, kiedy zachodzi potrzeba sprawdzenia kolizji promienia z animowaną siatką w danej pozycji (metoda `RayCollision_Bones`).

Wyliczone tablice macierzy kości są buforowane wewnętrznie w strukturach typu `BoneMatrixCacheEntry`. Pamiętana jest ograniczonej długości lista ostatnio używanych takich wpisów, a ich wymiana następuje wg polityki LRU (ang. *Last Recently Used*). Ponowne użycie wpisu przesuwa go z powrotem na koniec listy. Dopasowanie podanych parametrów określających wybraną pozycję animacji do wpisów przechowujących zapamiętane tablice macierzy kości może się odbywać z pewną tolerancją. Jej zwiększanie powoduje, że rzadziej zachodzi potrzeba wyliczania nowej tablicy macierzy, a co za tym idzie większa jest wydajność pracy silnika. Z drugiej strony jednak animacja jest wtedy mniej płynna.

**Moduł Multishader** **Shader** [67] to program wykonywany przez procesor karty graficznej (GPU), pisany w specjalnym języku (dawniej był to dedykowany assembler, obecnie używany jest język wysokiego poziomu podobny do C — Cg, GLSL lub HLSL). Nie daje on takich możliwości jak języki programowania CPU (nie posiada zmiennych globalnych, wskaźników, klas itp., a warunki, pętle i wywołania zostały wprowadzone dopiero w nowych wersjach). Wykonywany jest za to z prędkością nieosiągalną nawet na najlepszych dostępnych obecnie procesorach CPU. Vertex Shader wykonuje się dla każdego przetwarzanego wierzchołka, a Pixel Shader dla każdego rysowanego piksela

obrazu, w każdej klatce. Użycie shaderów nazywane jest potokiem programowalnym (ang. *Programmable Pipeline*) i zastępuje ustawienia dostępne na starych generacjach kart graficznych w ramach tzw. potoku predefiniowanego (ang. *Fixed Function Pipeline*). Dlatego na shadery można patrzeć jak na bardziej elastyczną metodę określania sposobu, w jaki karta przetwarza dane, pozwalającą na zapisanie dowolnej formuły matematycznej.

Pisząc zaawansowany silnik graficzny programista napotyka w tym miejscu pewien problem. Zadaniem shadera jest dokonanie wszelkich przekształceń jakie mają się odbywać na renderowanych wierzchołkach i pikselach — w tym animację szkieletową, oświetlenie, cień, teksturowanie itd. Każda z tych czynności może być konfigurowana na różne sposoby. Shaderów nie można jednak ze sobą łączyć — jednocześnie uaktywniony może być tylko jeden Vertex Shader i jeden Pixel Shader i musi on wykonywać wszystkie wymagane czynności.

Stąd powstaje potrzeba utworzenia wielu różnych shaderów dostosowanych do poszczególnych kombinacji czynności, które mają przeprowadzać (np. oświetlenie światłem punktowym, brak cienia, materiał nie posiadający odblasku). Duże możliwości silnika szybko prowadzą jednak do eksplozji kombinatorycznej tych ustawień czyniąc niemożliwym ręczne przygotowanie, a nawet automatyczne wygenerowanie w rozsądnym czasie shaderów dla wszystkich takich kombinacji. Powstało więc kilka metod poradzenia sobie z tym problemem [68].

Podejście naiwne zakłada przygotowanie jednego lub kilku shaderów posiadających pełny zakres możliwości i tak zaprojektowanych, aby niektóre z obliczeń można było pomijać podając odpowiednie wartości jako stałe. Na przykład jeśli kolor wyjściowy może być dodatkowo mnożony przez pewien określony kolor stały $C$, to podanie jako wartość $C$ koloru białego zaowocuje otrzymaniem niezmienionego koloru, tak jakby ta funkcja nie była w shaderze aktywna. Nadal jednak dodatkowe mnożenie koloru przez stałą byłoby wykonywane spowalniając niepotrzebnie renderowanie.

Nieco bardziej zaawansowane podejście zakłada użycia dostępnych w nowych wersjach *Shader Model* instrukcji warunkowych (ang. *Branching*). Jako stałe podawać można wartości logiczne oznaczające włączanie lub wyłączanie poszczególnych możliwości shadera. Sprawdzanie tych wartości pozwala na warunkowe wyłączanie fragmentów kodu shadera. Zdaniem autora nie warto jednak marnować czasu na sprawdzanie takiego warunku w kodzie wykonywanym dla każdego wierzchołka czy piksela, jeśli cały obiekt może być wyrenderowany z użyciem shadera w wersji całkowicie pozbawionej danego fragmentu kodu.

Najbardziej zaawansowane podejście zakłada kompilowanie na żądanie (przy pierwszym użyciu) wersji shadera posiadającej tylko potrzebne funkcje. W celu budowania kodu źródłowego takiego shadera zastosować można jeden z dwóch modeli [68]:

- **Model addytywny** [69] polega na składaniu kodu shadera z pewnych fragmentów, przy czym wyjścia jednych są łączone z wejściami innych. Wymaga to opra-

cowania sposobu opisywania tych fragmentów oraz ich łączenia. Może się to odbywać w specjalnym tekstowym formacie pliku bądź w sposób graficzny — za pomocą spotykanego w najbardziej zaawansowanych silnikach edytora materiałów, w którym użytkownik może łączyć prostokątne bloczki reprezentujące fragmenty shadera za pomocą myszki.

- **Model subtraktywny** polega na przygotowaniu jednego długiego kodu shadera posiadającego wszystkie przewidywane możliwości i wplecenia do niego dyrektyw preprocesora kompilacji warunkowej takich jak `#ifdef`. Język shaderów wysokiego poziomu HLSL posiada bowiem preprocesor bardzo podobny do tego w C/C++. Dzięki nim kompilator dla konkretnego shadera może wybrać tylko pewne fragmenty kodu poprzez zdefiniowanie używanych do kompilacji makr preprocesora.

Autor wybrał podejście subtraktywne. Podejście to czyni shader prostszym do napisania, ale równocześnie dużo trudniejszym do konserwacji. Kod głównego shadera w opisanym w tej pracy silniku liczy 907 wierszy. Shader ten jest opisany w rozdz. 3.2.

Moduł Multishader, którego kod zgromadzony jest w plikach `Framework\Multishader.hpp` i `Framework\Multishader.cpp`, definiuje klasę `Multishader`. Jest to rodzaj zasobu, który reprezentuje zbiór shaderów kompilowanych na żądanie z podanymi ustawieniami, powstających ze wspólnego kodu źródłowego w języku HLSL (wczytywanego z pliku FX). Owe ustawienia to tablica wartości całkowitoliczbowych ustawianych jako makra preprocesora podczas kompilacji shadera. Na podstawie tych wartości, składanych w sposób bitowy, powstaje tzw. hash — pojedyncza liczba 32–bitowa, która jednoznacznie identyfikuje dany shader. Jest ona używana do sprawdzenia, czy shader o podanych ustawieniach jest już wczytany do pamięci. Jeśli nie jest, klasa sprawdza, czy był wcześniej kompilowany i zapisany w wynikowej postaci binarnej do pliku w katalogu tymczasowym. Jeśli tak, wczytuje ten plik. Jeśli nie znajdzie takiego pliku, dopiero wówczas kompiluje shader ze źródła.

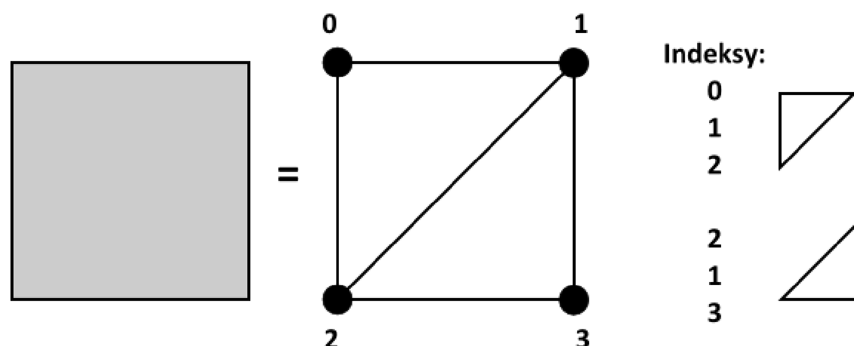
**Moduł Gfx2D** Kod znajdujący się w plikach `Framework\Gfx2D.hpp` i `Framework\Gfx2D.cpp`, zgromadzony w przestrzeni nazw `gfx2d`, to moduł służący do rysowania grafiki dwuwymiarowej za pomocą Direct3D. Możliwości takie są potrzebne w każdej aplikacji graficznej. W przypadku gier dwuwymiarowych za pomocą tego modułu można rysować całą grafikę. Jednak nawet w przypadku gier i programów trójwymiarowych, zawsze zachodzi potrzeba rysowania pewnych elementów płaskich, jak choćby kontrolek interfejsu użytkownika.

Biblioteki graficzne mogą być zaprojektowane na dwa sposoby, przy czym przedstawiony tu podział dotyczy równie dobrze grafiki 2D, jak i 3D. Pierwszy model to dostarczenie funkcji rysujących, które za każdym razem muszą zostać wywołane w celu

odrysowania poszczególnych elementów graficznych na ekranie. Takie API posiada np. Windows GDI, jak również Direct3D. Drugi model polega na przechowywaniu przez bibliotekę kolekcji obiektów graficznych, na których użytkownik może manipulować dodając je i usuwając, zmieniając ich pozycje, wymiary, kolor i inne parametry. Biblioteka sama zajmuje się każdorazowym odrysowaniem wszystkich pamiętanych obiektów.

Opisywany tu moduł grafiki 2D używa pierwszego modelu. Udostępnia on zmienną globalną `g_Canvas` typu `Canvas`, która posiada metody służące do rysowania elementów graficznych 2D: prostokątów wypełnionych jednolitym kolorem, prostokątów wypełnionych teksturą (*spritów* — p. niżej) oraz tekstu. Służą do tego metody odpowiednio `DrawRect` oraz `DrawText_`. Ponadto między wywołaniami rysującymi można przestawiać parametry rysowania za pomocą metod: `SetColor`, która ustawia bieżący kolor, `SetSprite`, która ustawia bieżący *sprite* oraz `SetFont`, która ustawia bieżącą czcionkę.

Oprócz parametrów rysowania ustawianych w normalny sposób (np. użytkownik ustawia kolor biały, rysuje białe prostokąty, zmienia kolor na czerwony, rysuje czerwone prostokąty), klasa posiada parametry odkładane na stos. Są to: `PushAlpha` i `PopAlpha` ustalające stopień półprzezroczystości, `PushTranslation` i `PopTranslation` ustalające przesunięcie oraz `PushClipRect` i `PopClipRect` ustalające prostokąt obcinania. Rysowanie prostokątów, jak również pojedynczych znaków tekstu, odbywa się poprzez rysowanie tzw. *quadów*, czyli par trójkątów (patrz rys. 2.4). Klasa nie wspiera przekształceń takich jak obroty, co było świadomą decyzją projektową, ponieważ ich wprowadzenie znacznie skomplikowałoby algorytm przycinania i tym samym uczyniłoby go mniej wydajnym.

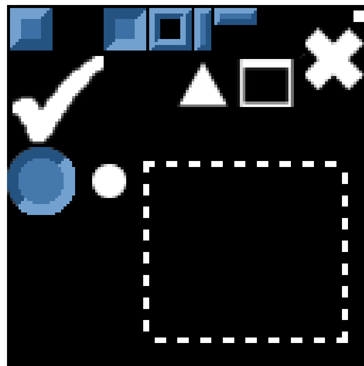



Rys. 2.4. Prostokąty 2D rysowane z użyciem Direct3D powstają z dwóch trójkątów utworzonych za pomocą 4 wierzchołków i 6 indeksów.

Współczesne karty graficzne są zbudowane w taki sposób, że zdolne są do rysowania ogromnej ilości geometrii, ale w celu uzyskania ich pełnej wydajności potrzebują dostawać tą geometrię w dużych porcjach (ang. *Batch*). Tymczasem rysowanie z użyciem klasy `Canvas` odbywa się za pomocą pojedynczych prostokątów. Aby przyspieszyć ten proces, klasa wewnętrznie buforuje wywołania rysujące starając się zbierać zakolejkowane w pamięci tak dużo wywołań rysowania prostokątów, jak to tylko moż-

liwe i wysyłać je do karty w graficznej w jednej porcji. Oczywiście rysowanie wielu prostokątów na raz musi się odbywać z użyciem tych samym ustawień (jak tekstura, shader itd.). Dlatego każda zmiana *sprita* powoduje opróżnienie kolejki. Toteż uzyskiwana wydajność rysowania zależy w dużej mierze od sposobu, w jaki programista używa tego modułu. Na przykład jeśli do narysowania jest grupa przycisków interfejsu użytkownika i są one rysowane po kolei, gdzie na przemian następują wywołania rysowania tła przycisku i umieszczonego na nim tekstu, wtedy każde takie wywołanie zmusza klasę `Canvas` do opróżnienia kolejki. Jeśliby natomiast najpierw narysowane zostały tła wszystkich przycisków, a następnie teksty wszystkich przycisków, klasa może lepiej zoptymalizować te wywołania.

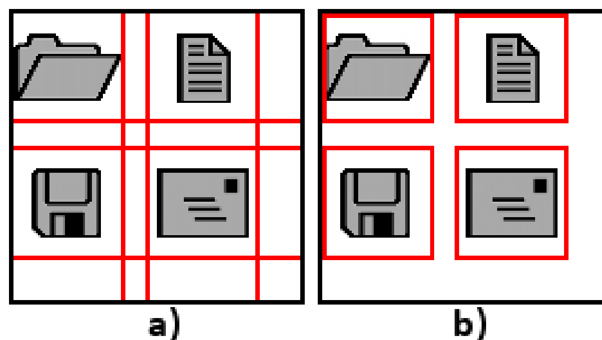
Częstą praktyką w grafice dwuwymiarowej jest umieszczanie na jednej teksturze wielu różnych elementów, tak jak to pokazuje rys. 2.5. Zachodzi potrzeba wyznaczenia prostokątnych obszarów na tej teksturze zawierających poszczególne elementy. W tym celu moduł wprowadza pojęcie *sprite* (czyt. „sprajt"), któremu w kodzie odpowiada klasa `Sprite`. Zawiera ona odniesienie do zasobu tekstury, efektu używanego do jej rysowania oraz dane na temat tych prostokątów. Sama nie jest zasobem, ponieważ zasoby nie mogą być od siebie zależne. Zamiast tego moduł grafiki 2D przechowuje kolekcję *spritów* w obiekcie globalnym `g_SpriteRepository` dając do nich dostęp poprzez nazwy. Definicje *spritów* można wczytywać z pliku tekstowego w specjalnym formacie bądź tworzyć z poziomu kodu.



Rys. 2.5. Tekstura z elementami grafiki interfejsu użytkownika używana w przykładowym kodzie silnika dołączonym do pracy.

Metoda `SetSprite` klasy `Canvas` ustawia aktywny *sprite* używany do rysowania prostokątów. Podczas rysowania konkretnego prostokąta metodą `DrawRect` podać należy numer elementu, który ma zostać wybrany do narysowania z aktywnego *sprita*. Numer ten może być jednego z trzech rodzajów. Stała `INDEX_ALL` to specjalna wartość oznaczająca cały obszar tekstury używanej przez *sprite*. Element prosty, używany poprzez podanie po prostu wartości jego indeksu, oznacza prostokątny obszar tekstury wyznaczony w definicji sprita na jeden z dwóch sposobów — jako macierz lub jako prostokąty o dowolnie określonych współrzędnych. Elementy tekstury pokazanej na rys. 2.6 mogą być wyznaczone za pomocą macierzy (której siatka jest zaznaczona

czerwonymi liniami w 2.6 a) za pomocą poniższej definicji:




Rys. 2.6. Sposoby wyznaczania elementów tekstury typu prostego w definicji *sprite'a* a) za pomocą macierzy, b) za pomocą prostokątów.

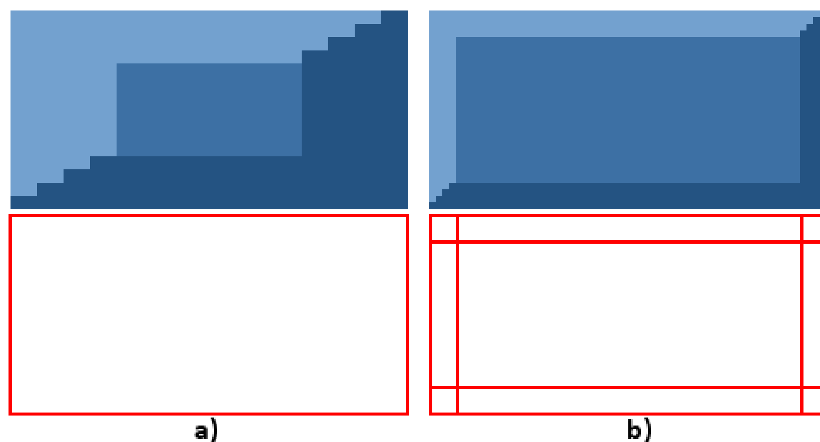
```
matrix {
  2     // Liczba kolumn
  48 48 // Szerokość i wysokość elementu
  12 0  // Odstęp i pozycja początkowa pozioma
  12 0  // Odstęp i pozycja początkowa pionowa
}
```

bądź za pomocą prostokątów (które zaznaczone są na czerwono na rysunku 2.6 b) za pomocą poniższej definicji:

```
rects {
  0 { left= 0 top= 0 width=48 height=48 }
  1 { left=60 top= 0 width=48 height=48 }
  2 { left= 0 top=60 width=48 height=48 }
  3 { left=60 top=60 width=48 height=48 }
}
```

Elementów graficznych zdolnych do rozszerzania się, takich jak okna czy przyciski, nie sposób jednak rysować estetycznie poprzez rozciąganie na ich powierzchni pojedynczego obrazu przedstawiającego tło takiego okna czy przycisku, co widać na rys. 2.7 a. Trzeba składać taki obraz z kilku połączonych elementów przedstawiających osobno rogi, osobno krawędzie i osobno środek wypełniający prostokąt, przy czym obraz przedstawiający krawędzie i środek może być, zależnie od konkretnego zastosowania, rozciągnięty lub wielokrotnie powtórzony. Dla uproszczenia rysowania takich konstrukcji moduł wspiera dodatkowo tzw. elementy złożone. Ich definicja odwołuje się do elementów prostych zdefiniowanych na jeden z opisanych wyżej sposobów określając osobno, które proste elementy mają posłużyć do rysowania rogów, które do rysowania krawędzi i który do rysowania wypełnienia. Dodatkowo elementy te mogą być obracane i odbijane, aby jeden element mógł posłużyć np. do narysowania wszystkich czterech rogów przycisku. Numer elementu złożonego podaje się do metody `DrawRect` poprzez użycie funkcji `ComplexElement`. Efekt narysowania przycisku zdefiniowanego jako element złożony przedstawia rys. 2.7 b).

Osobnym zagadnieniem jest funkcjonalność rysowania tekstu. Tekst renderowany z użyciem Direct3D musi powstawać, podobnie jak cała grafika, z oteksturowanych trójkątów. Dlatego każdy znak jest renderowany jako *quad* złożony z dwóch takich




Rys. 2.7. Rysowanie przycisku jako przykład elementu złożonego zbudowanego z elementów prostych wyznaczających osobno jego krawędzie, rogi i środek.

trójkątów. Tekstura przedstawiająca znaki jest przygotowana przez zewnętrzny program — Bitmap Font Generator (patrz rys. 2.8). Osobny plik tekstowy w specjalnym formacie, wyznaczający obszary zajmowane przez poszczególne znaki, jest również generowany przez ten program. Na jego podstawie klasa `Font` zawarta w module Res_d3d potrafi wyznaczać wymiary poszczególnych znaków, a dzięki nim szerokość całego tekstu, jak również automatycznie dzielić tekst na wiersze na granicy słowa i dodawać dodatkowe formatowania takie jak podkreślenia i przekreślenia.



Rys. 2.8. Tekstura ze znakami czcionki Arial Bold jako przykładowy efekt działania programu Bitmap Font Generator.

**Moduł GUI** **GUI** (ang. *Graphical User Interface* — graficzny interfejs użytkownika) oznacza zwykle zbiór okien i różnych kontrolek (takich jak przycisk, lista, pole edycyjne), którymi użytkownik może wygodnie manipulować za pomocą klawiatury i myszki. Nie każdy program czy gra potrzebuje zaawansowanego interfejsu użytkownika. Są jednak takie zastosowania (np. gry MMORPG), w których rozbudowane GUI jest konieczne. Równocześnie pisząc aplikację wykorzystującą do renderowania grafiki

Direct3D nie sposób wykorzystać standardowych kontrolek interfejsu użytkownika dostarczanych przez system operacyjny. Dlatego konieczne jest napisanie własnego systemu GUI.

System taki jest częścią kodu tej pracy zgromadzoną w plikach `Framework\GUI.hpp` i `Framework\GUI.cpp`. Jego elementy znajdują się w przestrzeni nazw `gui`. Podstawowym obiektem jest dostępny jako zmienna globalna `g_GuiManager`. Przechowuje on las kontrolek — trzy osobne drzewa zgrupowane na trzech warstwach (`LAYER_NORMAL` przeznaczona dla zwykłych okien, `LAYER_STAYONTOP` przeznaczona dla okien zawsze na wierzchu i `LAYER_VOLATILE` przeznaczona dla tymczasowych kontrolek takich jak menu albo listy rozwijalne). Każda kontrolka jest obiektem klasy dziedziczącej z `Control`. Obiekty klas dziedziczących z `CompositeControl` mogą posiadać swoje podkontrolki tworząc drzewo. Jest to wzorzec projektowy nazywany kompozytem (ang. *Composite*).

Oczywiste jest, że klasa kontrolki takiej jak przycisk jest jedna i jej kod nie jest odpowiedzialny za realizowanie każdej funkcji, którą ma wywoływać dany przycisk (np. „OK” czy „Anuluj” w oknie dialogowym). Zadaniem kontrolki przycisku jest jedynie wysłać komunikat o kliknięciu do pewnego ustalonego odbiorcy. Komunikaty takie najwygodniej jest realizować za pomocą tzw. wywołań zwrotnych (ang. *Callback*), nazywanych też delegatami (ang. *Delegate*), zdarzeniami (ang. *Event*) czy sygnałami i slotami (ang. *Signals*, *Slots*). W środowisku strukturalnym mogłyby do tego posłużyć zwykłe wskaźniki na funkcje. Przy programowaniu obiektowym w C++ powstaje jednak problem, bo język ten (w przeciwieństwie do innych języków programowania, jak D, Delphi, C#) nie posiada mechanizmu wskaźników na metody w takiej postaci jak potrzebne tutaj. Dlatego do zrealizowania zdarzeń wysyłanych przez kontrolki użyta została zewnętrzna biblioteka FastDelegate.

Poniższy listing prezentuje metody wirtualne definiowane przez klasę bazową `Control`.

```
virtual void OnDraw(const VEC2 &Translation);
virtual bool OnHitTest(const VEC2 &Pos);
virtual void GetDefaultSize(VEC2 *Out);
virtual void OnEnable();
virtual void OnDisable();
virtual void OnShow();
virtual void OnHide();
virtual void OnFocusEnter();
virtual void OnFocusLeave();
virtual void OnMouseEnter();
virtual void OnMouseLeave();
virtual void OnRectChange();
virtual void OnMouseMove(const VEC2 &Pos);
virtual void OnMouseButton(const VEC2 &Pos,
  frame::MOUSE_BUTTON Button, frame::MOUSE_ACTION Action);
virtual void OnMouseWheel(const VEC2 &Pos, float Delta);
virtual void OnDragEnter(Control *DragControl,
  DRAG_DATA_SHARED_PTR a_DragData);
virtual void OnDragLeave(Control *DragControl,
  DRAG_DATA_SHARED_PTR a_DragData);
```

```
virtual void OnDragOver(const VEC2 &Pos, Control *DragControl,
  DRAG_DATA_SHARED_PTR a_DragData);
virtual void OnDragDrop(const VEC2 &Pos, Control *DragControl,
  DRAG_DATA_SHARED_PTR a_DragData);
virtual void OnDragCancel(const VEC2 &Pos, Control *DragControl,
  DRAG_DATA_SHARED_PTR a_DragData);
virtual void OnDragFinished(bool Success, Control *DropControl,
  DRAG_DATA_SHARED_PTR a_DragData);
virtual bool OnKeyDown(uint4 Key);
virtual bool OnKeyUp(uint4 Key);
virtual bool OnChar(char Ch);
```

Kontrolka może być widzialna (ang. *Visible*) lub niewidzialna. Stan widzialności przechodzi też na podkontrolki. Dzięki temu można ukrywać całe okna. Kontrolka może być aktywna (ang. *Enabled*) lub nieaktywna (ang. *Disabled*). Stan aktywności również przechodzi na podkontrolki. Kontrolka niewidoczna lub nieaktywna nie może otrzymywać skupienia, a tym samym zdarzeń od klawiatury ani też od myszy.

Istnieje możliwość definiowania kontrolek o nieregularnych kształtach (np. okrągłym). Wymiary takiej kontrolki muszą być wówczas ustawione na jej prostokąt otaczający, a sprawdzanie czy dane miejsce znajduje się wewnątrz kontrolki realizowane jest przez metodę wirtualną `OnHitTest`.

Zdarzenia, jakie kontrolka dostaje od myszy obejmują wciśnięcie i zwolnienie przycisku (`OnMouseButton`), przesunięcie kursora (`OnMouseMove`) oraz obrócenie rolki (`OnMouseWheel`). Zdarzenia te kontrolka otrzymuje tylko jeśli kursor znajduje się nad jej obszarem lub jeśli jest ona w stanie przechwytywania myszy (ang. *Mouse Capture*). Stan przechwytywania włącza się, kiedy następuje wciśnięcie przycisku myszy nad obszarem kontrolki i trwa aż do zwolnienia tego przycisku. Oprócz tego kontrolka jest informowana o wejściu i wyjściu kursora myszy nad jej obszar wywołaniami metod odpowiednio `OnMouseEnter` i `OnMouseLeave`.

Dodatkowo obsługiwany jest mechanizm *Drag&Drop* — „Przeciągnij i upuść”, pozwalający na przeciąganie wirtualnych obiektów między kontrolkami za pomocą kursora myszy. Uczestniczy w tym procesie wiele różnych metod kontrolki, a przekazanie dowolnych danych między kontrolką źródłową a docelową odbywa się poprzez inteligentny wskaźnik zdefiniowany jako typ `DRAG_DATA_SHARED_PTR`.

Oprócz zdarzeń od myszy, system GUI obsługuje również wejście z klawiatury. W każdej chwili tylko jedna kontrolka ma tzw. skupienie (ang. *Focus*) i właśnie ona otrzymuje zdarzenia o naciśnięciach klawiszy. O otrzymaniu lub utracie skupienia informują kontrolkę metody `OnFocusEnter` i `OnFocusLeave`. Zdarzenia od klawiatury to `OnKeyDown` (oznacza wciśnięcie przycisku lub okresowe powtórzenie), `OnKeyUp` (oznacza zwolnienie przycisku) oraz `OnChar`. Ta ostatnia metoda, zamiast wirtualnego kodu klawisza, podaje wprowadzony znak. Jest to o tyle ważne, że klawisz nie jest jednoznaczny ze znakiem który wprowadza jego naciśnięcie. Dzięki otrzymywaniu osobnego zdarzenia odpowiedzialnego za wprowadzanie znaków kontrolki takie jak pole edycyjne nie muszą we własnym zakresie analizować czy np. wciśnięcie przycisku `A` oznacza wprowadzenie znaku `a`, `A` czy może `ą` na podstawie stanu klawiszy

`Shift`, `Alt` i `CapsLock`. Zajmuje się tym sam system operacyjny uwzględniając bieżące ustawienia językowe i układ klawiatury.

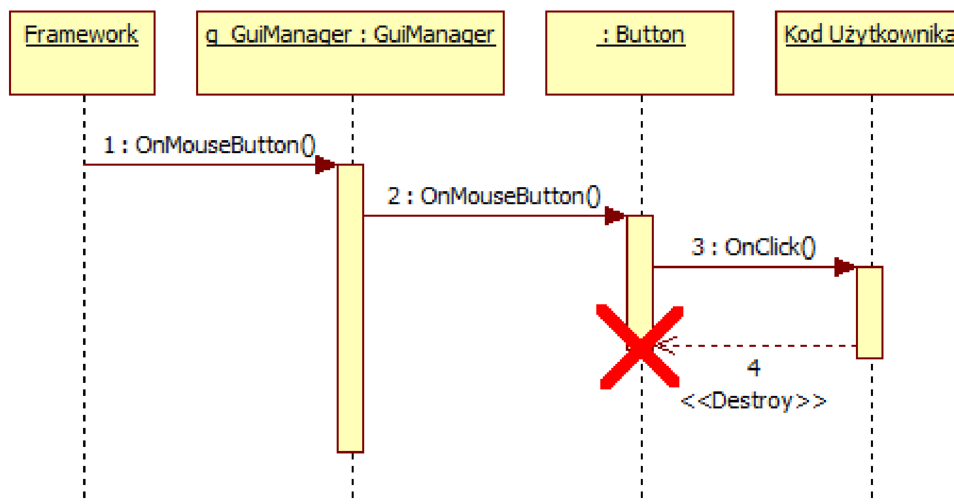
System GUI obsługuje też kursor. Rysowanie kursora należy do użytkownika, moduł definiuje jedynie pustą klasę bazową `Cursor`. To zapewnia maksymalną elastyczność. Każda kontrolka może mieć przypisany swój aktualny kursor, a system GUI udostępnia do odczytania w każdej chwili informację mówiącą, jaki kursor i w jakiej pozycji powinien być widoczny na ekranie. W przypadku kiedy trwa przeciąganie *Draw&Drop*, kursorem jest obiekt przeciągany, jako że klasa `DragData` jest pochodna od `Cursor`. Dzięki temu zamiast kursora w takiej sytuacji pokazywany może być przeciągany obiekt, co bywa przydatne np. kiedy w grze RPG użytkownik przekłada przedmioty w oknie ekwipunku.

Kolejna możliwość systemu GUI to wyświetlanie dymków z podpowiedzią (ang. *Hint*). Działają podobnie do kursorów. Ich rysowanie również leży w gestii użytkownika, a moduł definiuje jedynie pustą klasę bazową `Hint` oraz `PopupHint`. Każda kontrolka może mieć przypisaną do siebie podpowiedź. Istnieje możliwość odczytania w każdej chwili z systemu GUI, jaka podpowiedź i w jakim miejscu powinna być aktualnie pokazywana. Ponadto użycie klasy `PopupHint` pozwala na automatyczne obsługiwanie podpowiedzi „wyskakujących" w miejscu kursora po określonym czasie od jego zatrzymania nad obszarem danej kontrolki.

Poważnym problemem, jaki pojawił się podczas implementacji systemu GUI, było usuwanie kontrolek. Łatwo mogłoby dojść do sytuacji pokazanej na rys. 2.9. Jeśli na przykład użytkownik kliknął myszką na przycisk, szkielet przesyła odpowiednie zdarzenie do systemu GUI, który przesyła je z kolei do odpowiedniej kontrolki, nad którą znajduje się kursor myszy. Ta kontrolka, jako przycisk, generuje w reakcji na kliknięcie wywołanie zdarzenia o wciśnięciu przycisku. Jeśli to jest przycisk „OK" potwierdzający okno dialogowe, jego kliknięcie powinno spowodować zamknięcie i zwolnienie z pamięci całego okna, w tym także tego przycisku. Jednak kod klienta, który spróbowałby to zrobić, usunąłby obiekt będąc tak naprawdę w czasie wykonywania jednej z jego metod! Samo w sobie nie jest to błędem, ale jeśli jakiś kod w systemie GUI próbowałby dalej korzystać z tego obiektu, nieuchronnie doprowadziłoby to do zamknięcia całego programu z błędem ochrony pamięci. Dlatego autor zdecydował się na opóźnianie wywołań wszelkich zdarzeń wysyłanych przez system GUI do kontrolek (tych z listingu powyżej) poprzez umieszczanie ich w specjalnej kolejce.

Pliki `Framework\GUI_Controls.hpp` i `Framework\GUI_Controls.cpp` dostarczają standardowej implementacji pewnego zbioru kontrolek.

- `Label` reprezentuje statyczny tekst, który służy tylko celom informacyjnym i nie odbiera żadnych zdarzeń od użytkownika.
- `Button` reprezentuje przycisk, który można „wciskać" za pomocą kliknięcia myszką, z pokazywanym na nim tekstem.
- `SpriteButton` reprezentuje przycisk z pokazywanym na nim obrazem.



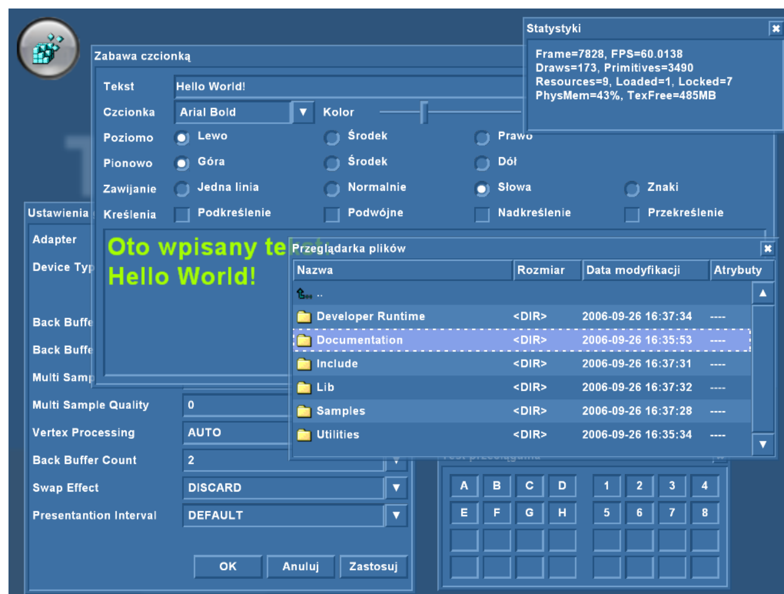

Rys. 2.9. Diagram sekwencji obrazujący problem z usuwaniem kontrolki w czasie wykonywania otrzymanego od niej zdarzenia.

- `CheckBox` reprezentuje pole opisane tekstem, które użytkownik może przestawiać w stan „zaznaczony"/„odznaczony" poprzez klikanie myszką.
- `RadioButton` reprezentuje pole posiadające opis tekstowy, które można zaznaczać, przy czym na raz może być zaznaczona tylko jedna kontrolka w grupie.
- `TrackBar` reprezentuje pasek, który można przesuwać zmieniając pewną wartość liczbową.
- `ScrollBar` reprezentuje pasek przewijania, przydatny na przykład do przewijania długich dokumentów tekstowych.
- `GroupBox` to kontrolka prezentująca prostokąt grupujący w sposób wizualny, jak i logiczny kontrolki umieszczone w jego wnętrzu.
- `TabControl` to zestaw zakładek, między którymi można się przełączać. Każda z nich wyświetla kartę, na której mogą być umieszczone inne kontrolki.
- `Menu` reprezentuje wyskakujące menu z poleceniami do wyboru.
- `Window` reprezentuje okno, które posiada pasek tytułowy, które można przesuwać i rozszerzać i które posiada w swoim wnętrzu inne kontrolki.
- `Edit` to najbardziej skomplikowana ze wszystkich kontrolek (jej kod źródłowy ma ponad 1200 linii). Zapewnia możliwość wprowadzania jednowierszowego tekstu wraz z przewijaniem długiego tekstu, pozycjonowaniem kursora, zaznaczaniem, obsługą schowka itp. funkcjami, które posiada również standardowa kontrolka systemowa tego typu.
- `List` to ogólna kontrolka prezentująca listę pozycji, zaprojektowana wg wzorca MVC (ang. *Model View Controller* — Model–Widok–Kontroler). Ta klasa bazowa stanowi kontroler (sposób obsługi komunikatów wejściowych), widok (sposób rysowania) realizuje jej klasa pochodna, a model (przechowywanie danych) realizuje osobny obiekt klasy pochodnej od *IListModel*.
- `TextComboBox` to lista rozwijalna, z której można wybrać tekst lub wpisać dowolny inny w dostępnym polu tekstowym.

- `ListComboBox` to lista rozwijalna bez możliwości wpisania własnego tekstu.

Ponadto osobny moduł, który stanowią pliki
`Framework\GUI_PropertyGridWindow.hpp`
i `Framework\GUI_PropertyGridWindow.cpp`, dostarcza rozbudowanej kontrolki `PropertyGridWindow`. Jest to okno prezentujące listę wartości różnego typu, przy czym do zmiany każdej z nich tworzone są osobne kontrolki zależnie od jej typu — np. do edycji wektora 3D powstają trzy pola edycyjne, do edycji koloru wraz z kanałem Alfa cztery suwaki i podgląd, do edycji łańcucha znaków powstaje pole tekstowe itd.

Zaimplementowany tu zestaw kontrolek nie wyczerpuje oczywiście wszystkich możliwości. Zależnie od konkretnych potrzeb może się okazać aż nadto rozbudowany lub też zbyt ubogi. Nie ma w nim np. drzewa (*TreeView*), pola do wprowadzania tekstu wielowierszowego (*Memo*) i innych kontrolek trudnych w implementacji. Zrzut ekranu z przykładowego programu demonstrującego możliwości opisanego tu systemu GUI prezentuje rys. 2.10.



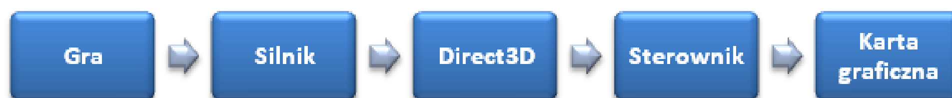

Rys. 2.10. **GUI TechDemo** — program demonstracyjny prezentujący możliwości systemu GUI (graficznego interfejsu użytkownika).

## 2.5. Architektura silnika

Właściwy kod silnika graficznego znajduje się w katalogu `Engine`. Do realizowania swoich funkcji wymaga jednak warstw niższych, dlatego nie sposób rozpatrywać go w oderwaniu od omówionych wcześniej modułów takich jak moduł matematyczny (patrz rozdz. 2.3) czy manager zasobów (patrz rozdz. 2.4).

Jak już zostało napisane we wstępie, silnik stanowi warstwę pośrednią między programem, a biblioteką graficzną taką jak DirectX. Ilustruje to rysunek 2.11. Biblioteka

graficzna udostępnia tylko funkcjonalność karty graficznej, a więc umożliwia renderowanie trójkątów oraz posługiwanie się zasobami niskiego poziomu takimi, jak bufor wierzchołków, bufor indeksów, tekstura czy shader. Silnik ukrywa takie szczegóły implementacyjne udostępniając obiekty reprezentujące pojęcia bardziej abstrakcyjne jak scena, kamera, światło czy model.




Rys. 2.11. Miejsce silnika w strukturze współczesnej gry komputerowej.

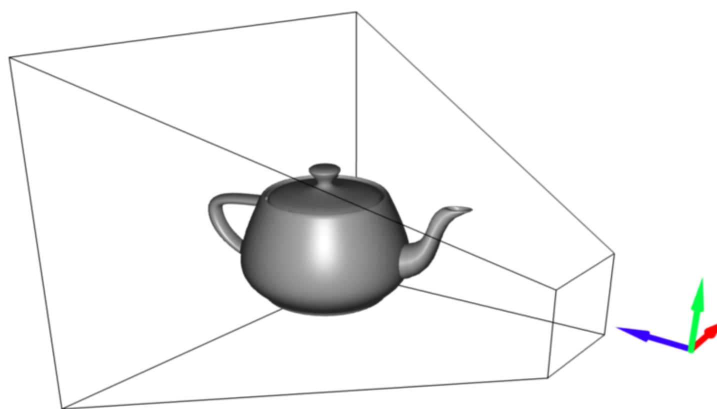
Ważnym założeniem przy projektowaniu tego silnika było, aby stanowił on zamkniętą bibliotekę enkapsulującą w swoim wnętrzu wszelkie szczegóły implementacyjne. Stoi to w kontraście do niektórych innych dostępnych w Internecie silników graficznych, które stanowią jakby tylko platformę oczekującą na rozszerzenie i zaimplementowanie różnych szczegółów. Autorowi zależało, aby tego silnika można było używać nie zajmując się problemami niskopoziomowymi i aby przez to był łatwy w użyciu. Na przykład użytkownik powinien operować jedynie na parametrach materiału bez wglądu w ich implementację, np. w kod shaderów. Zostało to osiągnięte kosztem możliwości łatwej rozbudowy, np. o nowe efekty materiałowe czy efekty postprocessingu.

**Moduł kamery** **Kamera** to pojęcie oznaczające „punkt widzenia”, z którego renderowana jest trójwymiarowa scena. Na niskim poziomie sprowadza się do dwóch macierzy:

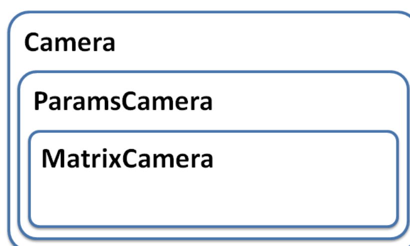
1. **Macierz widoku** (ang. *View Matrix*) dokonuje przekształcenia ze współrzędnych świata do współrzędnych kamery. Reprezentuje pozycję kamery w przestrzeni oraz jej orientację, w tym głównie kierunek patrzenia.
2. **Macierz rzutowania** (ang. *Projection Matrix*) dokonuje rzutowania perspektywicznego. Reprezentuje odległość bliskiej i dalekiej płaszczyzny obcinania (*Z–Near*, *Z–Far*), kąt widzenia (FOV — ang. *Field of View*) oraz stosunek szerokości do wysokości (ang. *Aspect Ratio*).

Ponadto obszar widoczny w kamerze można przedstawić jako *frustum* — ścięty ostrosłup o podstawie prostokąta (patrz rys. 2.12). Pobieranie jego kształtu jest potrzebne do testowania, czy obiekt jest widoczny z punktu widzenia danej kamery (ang. *Frustum Culling*). Jeśli nie jest, można pominąć jego rysowanie.

Zaprojektowanie funkcjonalnej, elastycznej i wygodnej klasy kamery nie jest proste, jeśli wziąć pod uwagę, że musi ona działać jak najwydajniej. Dlatego jej implementacja stosuje kilka nietypowych rozwiązań. Ogólna struktura klasy kamery pokazana



Rys. 2.12. Pojęcie kamery jako układu współrzędnych i frustuma.




Rys. 2.13. Struktura kamery.

jest na rysunku 2.13. Każda klasa wewnętrzna może istnieć samodzielnie, natomiast każda zewnętrzna zawiera w sobie też pokazane klasy wewnętrzne.

Klasa `MatrixCamera` reprezentuje tą najbardziej wewnętrzną, docelową część parametrów kamery. Obiekt tej klasy powstaje przez podanie macierzy widoku `View` i rzutowania `Proj`. Przechowuje oraz udostępnia do odczytu te macierze, jak również ich wersję połączoną `ViewProj` i ich odwrotności (`ViewInv`, `ProjInv`, `ViewProjInv`). Ponadto generuje na ich podstawie frustum opisany przez płaszczyzny (typu `FRUSTUM_PLANES`), opisany przez wierzchołki (typu `FRUSTUM_POINTS`) oraz AABB tego frustuma (typu `BOX`).

Klasa `ParamsCamera` reprezentuje kamerę sterowaną parametrami, na podstawie których wyznaczane są te dwie macierze. Obiekt tej klasy powstaje przez podanie: pozycji kamery `EyePos`, kierunku patrzenia `ForwardDir`, kierunku wskazującego górę `UpDir`, kąta widzenia pionowego `FovY`, stosunku szerokości do wysokości `Aspect` oraz odległości bliskiej i dalekiej płaszczyzny przycinania `ZNear`, `ZFar`. Oprócz przechowywania i udostępniania tych parametrów posiada pole typu `MatrixCamera`, do którego wpisuje macierze wygenerowane na ich podstawie. Potrafi także wyliczyć dodatkowe wektory potrzebne do niektórych operacji — wektor w prawo `RightDir` i „prawdziwy" wektor do góry `RealUpDir`, które razem z wektorem kierunku patrzenia tworzą bazę ortonormalną.

Klasa `Camera` reprezentuje kamerę sterowaną takimi parametrami, jakimi zwykle posługuje się użytkownik silnika 3D. Oprócz parametrów rzutowania (takich samych

jak w klasie powyżej) są to: pozycja kamery oraz jej orientacja. Orientacja może być opisana na wybrany spośród dwóch sposobów: za pomocą dwóch z kątów Eulera (`AngleY`, `AngleX`) lub za pomocą kwaterniona (`Orientation`). Ta pierwsza możliwość używana jest, kiedy użytkownik programu ma możliwość bezpośredniego sterowania kamerą za pomocą myszki. Ta druga natomiast przydatna jest do automatycznego sterowania kamerą, np. podczas scenek filmowych (ang. *Cutscene*), kiedy kamera płynnie przesuwa się i obraca za pomocą interpolacji po z góry ustalonej krzywej. Ponadto punkt widzenia kamery może być odsunięty względem jej wirtualnej „pozycji" o podaną odległość `CameraDist`, co pozwala łatwo zrealizować perspektywę TPP (ang. *Third Person Perspective* — perspektywa trzeciej osoby). Ustawienie tej wartości na 0 daje kamerę FPP (ang. *First Person Perspective* — perspektywa pierwszej osoby).

Klasa ta przechowuje w sobie obiekt typu `ParamsCamera`, wpisuje do niego parametry wyliczone na podstawie swoich danych i daje dostęp do tego obiektu. Dodatkową funkcją tej klasy jest umiejętność wyliczenia parametrów promienia (półprostej) we współrzędnych świata na podstawie podanej pozycji $(x, y)$, na przykład miejsca kliknięcia myszką na ekranie.

Parametry, które każda z tych klas potrafi wyliczać, są wyliczane przy pierwszym ich użyciu. To tzw. wartościowanie leniwe (ang. *Lazy Evaluation*). Dzięki niemu przykładowo niezależnie od tego, czy nastąpi najpierw 10 razy zmiana parametrów kamery i potem 1 raz odczytanie jej macierzy, czy 1 raz zmiana parametrów i 10 odczytań macierzy, macierze zostaną wyliczone na podstawie najnowszych ustawionych parametrów tylko raz.

Ponadto klasy kamery są przygotowane na ich używanie jako obiekty stałe (słowo kluczowe `const`). Na przykład przekazanie do jakiejś funkcji parametru typu `const ParamsCamera &` oznacza, że funkcja ta może odczytywać dane kamery, ale nie może zmieniać jej ustawień. Aby mimo modyfikatora `const` możliwe było leniwe wartościowanie, podlegające mu pola są oznaczone jako `mutable`.

**Moduł Engine** Jądro całego silnika stanowi kod znajdujący się w plikach `Engine\Engine.hpp` i `Engine\Engine.cpp`, zgromadzony w przestrzeni nazw `engine`. Podstawową klasą jest `Engine`, której jedyny, dostępny dla wszystkich obiekt zapamiętany jest w zmiennej globalne `g_Engine`. Reprezentuje on cały silnik graficzny. Zawiera kilka ustawień, które pozwalają na wyłączanie niektórych funkcji graficznych, jak oświetlenie czy cienie. Oprócz tego, jego główną rolą jest przechowywanie kolekcji scen, z których jedna jest wybrana jako aktywna.

Klasa `EngineServices` jest przeznaczona tylko do użytku wewnętrznego dla innych klas modułu Engine. Obiekt klasy `Engine` przechowuje i udostępnia pojedynczy obiekt tej klasy. Jego zadaniem jest dostarczanie zasobów pomocniczych potrzebnych w procesie renderowania, jak tekstury wykorzystywane w roli mapy cienia. Zajmuje się także ustawianiem stanów Direct3D i parametrów shadera.

**Scena**, czyli obiekt klasy `Scene`, reprezentuje „wirtualny świat" —- przestrzeń 3D, w której umieszczone są wszystkie obiekty graficzne. Jest najobszerniejszą klasą tego modułu. Przechowuje wskaźniki do zawartych w niej obiektów graficznych różnego rodzaju i zawiera procedury zarządzające ich wydajnym renderowaniem. Dodatkowo posiada możliwość sprawdzania trafienia (kolizji) promienia do zawartych w niej obiektów. Scena zawiera następujące obiekty i parametry:

- zbiór encji typu `Entity`, czyli podstawowych obiektów graficznych, Encje są przechowywane w swobodnym drzewie ósemkowym, opisanym szerzej w rozdz. 3.13,
- zbiór świateł typu `BaseLight`,
- zbiór kamer typu `Camera`, z których jedna ustawiona jest jako aktywna,
- kolekcja materiałów — pojedynczy obiekt typu `MaterialCollection`,
- zbiór efektów postprocessingu,
- obiekt mapy dla przestrzeni zamkniętych typu `QMap` oraz zbiór parametrów wyświetlania tej mapy (czy podlega oświetleniu, czy rzuca cień, czy otrzymuje cień),
- teran dla przestrzeni otwartych typu `TerrainRenderer`, który obejmuje wyświetlanie podłoża terenu, drzew, trawy i wody,
- obiekt nieba typu `BaseSky`,
- obiekt opadów atmosferycznych typu `Fall`,
- kolor światła otoczenia (ang. *Ambient Light*) typu `COLORF`,
- wektor oznaczający siłę i kierunek wiatru typu `VEC3`,
- parametry mgły — jej stan (czy jest włączona), kolor i odległość, od której się rozpoczyna.

**Encja** (ang. *Entity*) to podstawowy budulec sceny, dowolny obiekt potrafiący narysować się na ekranie i posiadający szereg parametrów, pośród których najważniejsze jest położenie w przestrzeni 3D. Z klasy bazowej `Entity` dziedziczy kilka innych klas rozpoznawanych za pomocą metody `GetType` i rzutowanych w dół, z których z kolei dziedziczą klasy realizujące konkretne rodzaje encji, opisane w następnym podrozdziale. Te typy to:

- klasa `MaterialEntity` (typ `TYPE_MATERIAL`) — obiekt, którego obraz rysowany jest z użyciem standardowego mechanizmu materiałów silnika. Posiada dodatkowe parametry:
  - `TeamColor` to kolor charakterystyczny dla danej encji, który może wchodzić w interakcję z materiałem,
  - `TextureMatrix` to macierz transformacji współrzędnych tekstury,
  - flagi mówiące czy encja podlega oświetleniu, czy rzuca cień i czy otrzymuje cień,
- klasa `CustomEntity` (typ `TYPE_CUSTOM`) — obiekt, którego obraz rysowany jest w niestandardowy sposób,
- klasa `HeatEntity` (typ `TYPE_HEAT`) — niewidoczny obiekt, który renderowany jest do kanału Alfa celem uzyskania efektu *Heat Haze*,

- klasa `TreeEntity` (typ `TYPE_TREE`) — drzewo.

Każda encja posiada następujące parametry:

- transformacja opisana za pomocą wektora translacji, kwaterniona orientacji oraz skalara dla skalowania równomiernego,
- encja nadrzędna i zbiór encji podrzędnych. Encje tworzą logiczne drzewo (które jednak nie ma nic wspólnego z drzewem ósemkowym, w którym są przechowywane dla optymalizacji). Dzięki temu ich transformacje są składane, co pozwala np. na automatyczne przemieszczanie i obracanie się modeli miecza, zbroi i hełmu wraz z modelem ich posiadacza. Istnieje również możliwość podłączenia encji do wybranej kości jej encji nadrzędnej (np. miecz obraca się razem z ręką posiadacza),
- stan widoczności — czy obiekt jest widoczny, czy ukryty,
- Tag — 32–bitowa liczba całkowita do dowolnego użytku.

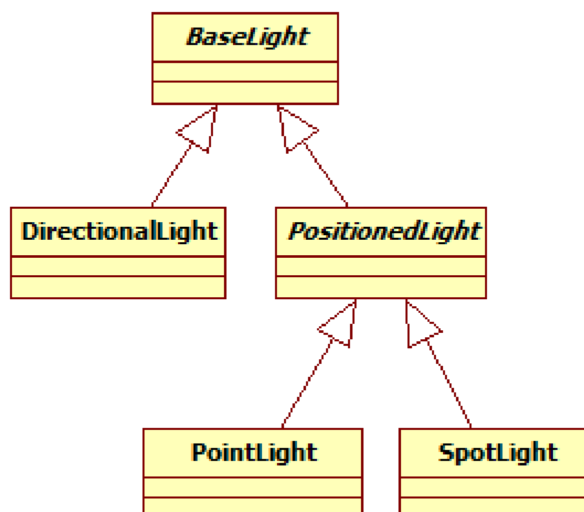
Ponadto istnieje możliwość otrzymania macierzy reprezentującej transformację encji i odwrotność tej macierzy, jak również promień sfery otaczającej. Autor zdecydował się na użycie brył otaczających w kształcie sfer ze względu na prostotę i szybkość ich użycia. Jest to ważne szczególnie podczas obracania się obiektów, kiedy to sfera zachowuje swój dotychczasowy kształt i wielkość, a innego rodzaju bryły otaczające wymagają dodatkowej obsługi. Klasa encji posiada także funkcję wirtualną testującą kolizję promienia z obiektem.

Rys. 2.14 pokazuje hierarchię klas świateł. `BaseLight` to klasa bazowa dla wszelkiego rodzaju świateł. Przechowuje ona dane wspólne dla każdego światła:

- kolor,
- aktywność — mówi czy światło jest włączone,
- flaga mówiąca czy światło rzuca cień,
- flaga mówiąca, czy światło powoduje odblask,
- flaga mówiąca, czy światło używa obliczeń *Half–Lambert*,
- współczynnik jasności dla miejsc znajdujących się w cieniu.

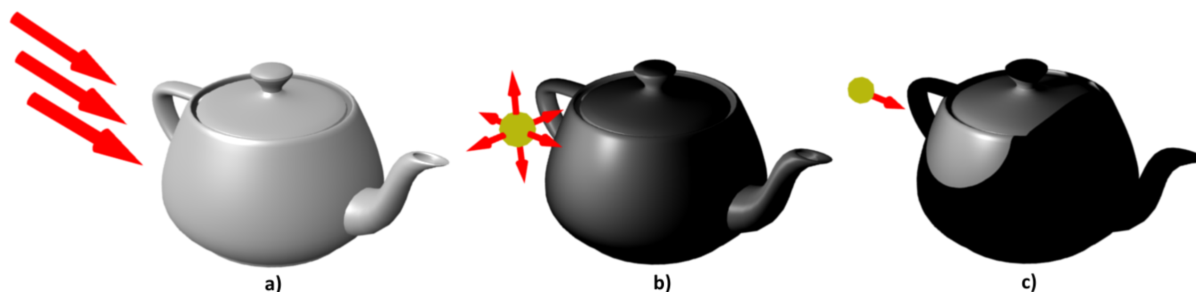
Klasa `PositionedLight` również jest abstrakcyjna. Stanowi bazę dla wszystkich rodzajów świateł, które mają określoną pozycję w przestrzeni. Oprócz przechowywania pozycji potrafi ona także zwrócić prostokąt wyznaczający zasięg działania światła w przestrzeni ekranu, używany w czasie renderowania do obcinania za pomocą *Scissor Test* przyspieszającego proces. Zwracana przy okazji wartość procentowa tego obszaru względem całej powierzchni ekranu może z kolei posłużyć do wyznaczenia poziomu szczegółowości, z jakim wyrenderowane ma być oświetlenie danym światłem, np. rozdzielczość używanej w jego przebiegu mapy cienia.

Światło kierunkowe [3], reprezentowane przez klasę `DirectionalLight`, to światło nie posiadające swojej pozycji w przestrzeni (patrz rys. 2.15 a). Rozchodzi się




Rys. 2.14. Diagram klas świateł.

ono równolegle oświetlając całą scenę, tak jakby pochodziło z nieskończenie odległego źródła. Jest najprostsze i najmniej kosztowne obliczeniowo. Nadaje się doskonale do modelowania światła słonecznego. Posiada kierunek padania (znormalizowany wektor). Dodatkowo przechowuje odległość, która wyznacza jak daleko od kamery sięga cień rzucany przez to światło. Ograniczenie tego zasięgu ma kluczowe znaczenie dla poprawy jakości cieni.




Rys. 2.15. Rodzaje świateł: a) światło kierunkowe, b) światło punktowe, c) światło latarki.

Światło punktowe [3], reprezentowane przez klasę `PointLight`, to światło rozchodzące się we wszystkich kierunkach z określonego punktu w przestrzeni (patrz rys. 2.15 b). Może modelować oświetlenie pochodzące np. od żarówki albo płomienia świecy. Posiada maksymalny zasięg, do którego jego intensywność zanika wg zależności kwadratowej. Posiada także odległość minimalną, od której rozpoczyna się działanie jego cienia (odległość ta nie może wynosić 0, co wynika z budowy macierzy rzutowania perspektywicznego). Używanie tego światła z włączonymi cieniami jest najbardziej kosztowne obliczeniowo, gdyż wymaga każdorazowego renderowania sceny osobno do każdej z 6 ścian sześciennej mapy cienia otaczającej światło ze wszystkich stron.

Trzeci rodzaj światła to SpotLight [3]. Reprezentuje ono światło rozchodzące się od określonego punktu w określonym kierunku, w zakresie ograniczonym przez pewien kąt do obszaru w kształcie stożka (patrz rys. 2.15 c). Światło to posiada pozycję, zasięg i minimalną odległość cienia (podobnie jak światło punktowe), a ponadto kierunek, kąt i flagę mówiącą, czy wraz ze wzrostem kąta jego intensywność ma płynnie zanikać.

W procesie renderowania elementów sceny istnieje kilka flag typu logicznego ('tak' lub 'nie'), które mogą przyspieszyć bądź spowolnić ten proces. Są to:

- Flaga SETTING_MATERIAL_SORT mówi, czy encje należy sortować wg materiału.
- Flaga SETTING_ENTITY_OCCLUSION_QUERY mówi, czy sprawdzać widoczność encji za pomocą zapytań *Occlusion Query*.
- Flaga SETTING_LIGHT_OCCLUSION_QUERY mówi, czy sprawdzać zasięg świateł za pomocą zapytań *Occlusion Query*.
- Flaga SETTING_TREE_FRUSTUM_CULLING mówi, czy wykonywać test przecięcia frustuma widzenia z bryłami otaczającymi drzew.

Na przykład jeśli scena zawiera encje rysowane każda innym materiałem, to ich sortowanie wg materiału nie przyniesie żadnych korzyści i jako dodatkowa operacja niepotrzebnie spowolni program. Jeśli natomiast do sceny dodane jest wiele encji rysowanych tym samym materiałem, to ich wyświetlanie jedna za drugą zminimalizuje liczbę potrzebnych zmian tekstury, shadera i innych ustawień Direct3D, co przyspieszy renderowanie. Podobnie jeśli wszystkie obiekty będące w zasięgu kamery faktycznie są widoczne, to wykonywanie dodatkowych zapytań sprzętowych o zasłanianie byłoby tylko stratą czasu. Jeśli natomiast jakieś skomplikowane obiekty są w zasięgu frustuma kamery, ale są zasłonięte np. przez ścianę, to wykonanie takiego zapytania zaoszczędzi bardzo wiele czasu, którego wymagałoby ich odrysowanie.

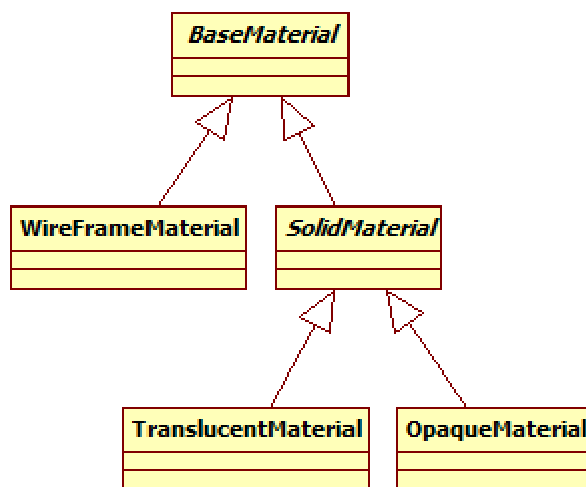
Zamiast ustawić wartości tych flag na stałe w kodzie silnika bądź też próbować wyliczać ich wartości optymalne dla danej sytuacji w jakiś sposób analityczny, autor przeznaczył do sterowania nimi klasę RunningOptimizer. Podstawowe założenie mówi, że klasa ta jest całkowicie „nieświadoma” znaczenia poszczególnych ustawień. Dla niej jest to tylko zbiór 4 zmiennych logicznych, którymi steruje. Wykonywana w każdej klatce metoda OnFrame wykonuje dwie czynności. Po pierwsze, analizuje czas trwania poprzedniej klatki. Po drugie, zwraca wartości zmiennych logicznych do zastosowania w bieżącej klatce.

Tylko tyle jest potrzebne, aby klasa automatycznie dobierała najoptymalniejsze w danej chwili wartości tych zmiennych. Zapamiętane są ich „aktualne” wartości i one są zwracane przez większość klatek pracy silnika. Raz na kilka klatek klasa „próbuje” jednak przestawić jedną z tych zmiennych na stan przeciwny, aby w następnej klatce sprawdzić, czy spowodowało to przyspieszenie renderowania. Po kilku takich pozytywnie zakończonych próbach „aktualny” stan danej zmiennej jest przestawiany na przeciwny.

Eksperymenty ze specjalnie przygotowanymi scenami dowiodły, że ten prosty i ogólny algorytm dobrze sprawdza się w praktyce. Klasa faktycznie dobiera parametry optymalne w danej chwili, choć robi to z kilkusekundowym opóźnieniem. Na przykład kiedy duża część sceny objętej zasięgiem kamery staje się zasłonięta przez umieszczoną tuż przed kamerą ścianę, po chwili uaktywnia się *Occlusion Query* znacznie przyspieszając renderowanie.

Można powiedzieć, że na wysokim poziomie grafikę 3D reprezentują dwa rodzaje danych: kształt i wygląd powierzchni. Kształt opisany jest siatką trójkątów. Materiał to pojęcie reprezentujące wygląd powierzchni i sposób, w jaki reaguje ona na światło. W jego skład wchodzą więc takie parametry, jak używana tekstura czy też intensywność odblasku.

Sposób obsługi materiałów w silniku może być różny. W najprostszym przypadku materiał może opisywać pojedyncza struktura złożona ze wszystkich dopuszczalnych parametrów. W najbardziej zaawansowanych silnikach istnieje osobny edytor materiałów, w którym sposób renderowania powierzchni można definiować bardzo elastycznie za pomocą edytora, używając połączonych ze sobą, wizualnych elementów. Autor zdecydował się na proste rozwiązanie, w których parametry materiału przechowuje jedna z klas, wybrana zależnie od konkretnego typu materiału. Hierarchię klas materiałów przedstawia rys. 2.16.




Rys. 2.16. Diagram klas materiałów.

`BaseMaterial` to abstrakcyjna klasa bazowa przechowująca parametry wspólne dla wszystkich materiałów:

- `Name` — nazwa,
- `TwoSided` — czy materiał jest dwustronny. Jeśli nie jest, włączony zostaje *Backface Culling*,
- `CollisionType` — maska bitowa określająca typ kolizji (patrz p. 2.5).

Klasa `WireFrameMaterial` reprezentuje materiał rysowany w trybie szkieletowym — jako same krawędzie trójkątów. Posiada następujące dodatkowe parametry:

- `Color` — kolor,
- `BlendMode` — tryb alfa–blendingu (zwykły, addytywny lub subtraktywny),
- `ColorMode` — skąd pobierany ma być kolor (z materiału, z `TeamColor` encji lub z iloczynu obydwu).

`SolidMaterial` to abstrakcyjna klasa bazowa reprezentująca wszystkie te materiały, które są rysowane z wypełnieniem trójkątów. Jej parametry to:

- `DiffuseTextureName` — nazwa zasobu podstawowej tekstury zawierającej kolor,
- `DiffuseColor` — kolor używany, jeśli nie jest podana tekstura,
- `EmissiveTextureName` — nazwa zasobu tekstury zawierającej miejsca „emitujące” światło. Encja nie jest tak naprawdę źródłem światła, ale te miejsca rysowane są w sposób niepodlegający oświetleniu, co pozwala przedstawić np. diody świecące na panelu komputera,
- `EnvironmentalTextureName` — nazwa zasobu tekstury sześciennej używanej do mapowania środowiskowego (ang. *Environmental Mapping*), adresowanej wektorem kierunku do kamery odbitym od powierzchni,
- `TextureAnimation` — wartość logiczna mówiąca, czy współrzędne tekstury podlegają transformacji przez macierz `TextureMatrix` encji. Jej użycie pozwala np. na przesuwanie czy rozciąganie tekstury na powierzchni obiektu,
- `ColorMode` — wartość wyliczeniowa określająca, czy kolor podstawowy ma być brany z materiału, z iloczynu materiału przez kolor encji `TeamColor` bądź też z interpolacji między nimi za pomocą parametru pobieranego z kanału Alfa podstawowej tekstury. To ostatnie ustawienie pozwala zaznaczyć na teksturze wybrane miejsca, które mają podlegać przekolorowaniu wg koloru danej encji. Dzięki temu można rysować tym samym materiałem np. postacie żołnierzy dwóch wrogich armii, które różnią się kolorem hełmu (stąd nazwa „TeamColor”).
- `FresnelColor` i `FresnelPower` to kolor i wykładnik określające parametry efektu *Fresnel Term*.

Klasa `TranslucentMaterial` reprezentuje materiał półprzezroczysty. Materiały półprzezroczyste są w opisywanym silniku oddzielone od całkowicie nieprzezroczystych, ponieważ nie mogą podlegać oświetleniu. Ta trudna decyzja projektowa podyktowana została problemem opisanym w p. 3.1. Klasa przechowuje następujące dodatkowe parametry:

- `BlendMode` — tryb alfa–blendingu,
- `AlphaMode` — skąd pobierana ma być wartość przezroczystości Alfa (z materiału, z `TeamColor` encji lub z iloczynu obydwu).

Klasa `OpaqueMaterial` reprezentuje najczęściej używany, całkowicie nieprzezroczysty materiał, który może podlegać oświetleniu. Jego dodatkowe parametry to:

- `AlphaTesting` — wartość graniczna Alfa. Jeśli niezerowa, powoduje uaktywnienie testu alfa (ang. *Alpha–Testing*),
- `HalfLambert` — czy oświetlenie liczone ma być metodą *Half–Lambert*,
- `PerPixel` — wartość logiczna określająca, czy oświetlenie ma być liczone dla wierzchołków i interpolowane na powierzchni trójkątów (metoda działająca szybciej, ale wyglądająca gorzej), czy dla poszczególnych pikseli. Warto dodać, że większość silników udostępnia tylko jeden z tych dwóch trybów,
- `NormalTextureName` — nazwa zasobu tekstury przechowującej mapę normalnych (ang. *Normal Map*),
- `SpecularMode` — tryb odblasku (ang. *Specular*): brak, zwykły odblask lub oświetlenie anizotropowe (ang. *Anisotropic Lighting*),
- `SpecularColor` — kolor własny odblasku. Jest mnożony przez kolor światła. Na kolor odblasku nie ma wpływu kolor podstawowy materiału,
- `SpecularPower` — wykładnik potęgi używanej do liczenia odblasku,
- `GlossMapping` — wartość logiczna określająca, czy intensywność odblasku ma być mnożona przez wartość Alfa pobieraną z tekstury podstawowej. Dzięki tej funkcji możliwe jest wyznaczanie na powierzchni modelu tylko wybranych fragmentów jako odblaskowe.

Jak widać, kanał Alfa tekstury podstawowej może posłużyć do różnych celów, zależnie od typu materiału i jego ustawień:

- W materiale półprzezroczystym typu `TranslucentMaterial` wyznacza stopień półprzezroczystości.
- Jeśli `AlphaTesting` $> 0$, wyznacza miejsca przezroczyste odrzucane za pomocą testu Alfa.
- Jeśli `ColorMode` jest równy `COLOR_LERP_ALPHA`, interpoluje między kolorem tekstury, a kolorem `TeamColor` encji.
- Jeśli `GlossMapping` jest równy `true`, opisuje intensywność odblasku.

Obliczenia kolizji, czyli przecięć geometrycznych między bryłami w 3D są w programowaniu gier uznawane za część kodu przeznaczonego do obliczeń fizyki. Opisany tu program, jako silnik czysto graficzny, nie posiada więc zaawansowanych kolizji ani żadnej fizyki ciała sztywnego. Do obliczeń fizycznych wykorzystuje się zwykle gotowe silniki, takie jak ODE, Newton, PhysX czy Havok.

Tematu obliczeń kolizji nie sposób jednak do końca oddzielić od grafiki. W niektórych przypadkach do tych obliczeń potrzebne są te same dane i procedury, które wykorzystywane są do renderowania grafiki. Na przykład w grze typu FPS (ang. *First Person Shooter* — strzelanka z perspektywy pierwszej osoby) obliczenia trafienia kulą z pistoletu musi być dokonane dokładnie, a więc z pojedynczymi trójkątami obiektów ustawionymi w aktualnej pozycji, być może sterowanej przez animację szkieletową.

Dlatego opisywany silnik posiada funkcję służącą do liczenia kolizji promienia z zawartością sceny. Ponieważ nie każdy obiekt musi reagować na każdy test kolizji, wprowadzone zostało pojęcie typu kolizji `CollisionType`. Jest to wartość typu `uint` interpretowana jako maska bitowa. Domyślne flagi przeznaczone do niej to: `COLLISION_PHYSICAL` oznaczająca kolizję fizyczną, `COLLISION_OPTICAL` oznaczająca kolizję optyczną i `COLLISION_BOTH` oznaczająca obydwa rodzaje kolizji. Każdy materiał przechowuje typ kolizji, na który reaguje. Dzięki typom kolizji można utworzyć takie obiekty, jak np. obiekt niewidzialny i przepuszczający światło lasera, ale reagujący na uderzenie albo hologram, który nie przepuszcza światła, ale pozwala obiektom przez niego przechodzić.

Klasa sceny dostarcza metodę o nagłówku:

```
COLLISION_RESULT RayCollision(COLLISION_TYPE CollisionType,
  const VEC3 &RayOrig, const VEC3 &RayDir,
  float *OutT, Entity **OutEntity);
```

Zwraca ona rodzaj obiektu, z którym nastąpiła kolizja: `COLLISION_RESULT_NONE` to brak kolizji, `COLLISION_RESULT_MAP` to kolizja z mapą, `COLLISION_RESULT_TERRAIN` to kolizja z terenem, `COLLISION_RESULT_ENTITY` to kolizja z encją. Metoda zwraca też przez parametry wskaźnikowe encję, z którą nastąpiła kolizja oraz odległość kolizji, wyrażona w wielokrotnościach długości wektora `RayDir`.

Implementacja tej kolizji opiera się na metodzie abstrakcyjnej zdefiniowanej w klasie bazowej encji:

```
virtual bool RayCollision(COLLISION_TYPE Type,
  const VEC3 &RayOrig, const VEC3 &RayDir, float *OutT) = 0;
```

Zadaniem klasy pochodnej jest dostarczyć jej implementacji, która oblicza precyzyjną kolizję promienia z kształtem obiektu danego rodzaju. Scena, przed wywołaniem tej metody dla danej encji, upewnia się najpierw czy promień koliduje ze sferą otaczającą tą encję, a także przekształca parametry promienia z globalnego układu współrzędnych świata do układu lokalnego danej encji mnożąc je przez odwrotność macierzy przekształcenia tej encji.

**Klasy encji** Kod zgromadzony w plikach `Engine\Entities.hpp` i `Engine\Entities.cpp` dostarcza klas implementujących różne rodzaje encji, pochodnych od klasy `Entity`.

Klasa `res::QMesh` jest zasobem przechowującym siatkę wczytaną z pliku QMSH. Klasa `QMeshEntity` jest encją przechowującą wskaźnik do takiego zasobu. O ile sam zasób siatki udostępnia jedynie jej dane i metadane oraz potrafi obliczyć macierze kości dla podanej animacji i czasu, o tyle encja zajmuje się odrysowaniem tej siatki oraz pamięta stan jej animacji. Klasa `QMeshEntity` jest więc podstawową klasą obiektu graficznego, reprezentującą instancję siatki ustawioną na scenie.

Jeśli siatka posiada szkielet, w każdej chwili odtwarzana może być zero, jedna lub dwie animacje. Do wyłączenia animacji służy metoda `ResetAnimation`. Do uruchomienia animacji służy metoda `SetAnimation`. Możliwe jest zakolejkowanie jednej animacji do otworzenia po zakończeniu bieżącej za pomocą metody `QueueAnimation`. Możliwe jest też płynne przejście (interpolacja) między animacją bieżącą i nową. Służy do tego metoda `BlendAnimation`. To bardzo ważne, bo dzięki tej funkcjonalności możliwe jest np. po puszczeniu klawisza klawiatury przerwanie animacji „Idzie" w dowolnym momencie i płynne przejście do animacji spoczynkowej "Stoi" bez nagłego przeskoku. Dzięki podaniu dodatkowych flag bitowych typu `ANIM_MODE` możliwe jest zapętlenie animacji, a także jej odtwarzanie wstecz lub na przemian w obydwie strony. Ponadto sterować można fazą (czasem początkowym), prędkością i bieżącym czasem animacji.

Siatka QMSH składa się z fragmentów, a każdy ma zapisaną nazwę materiału. Klasa `QMeshEntity` automatycznie pobiera materiały poszczególnych fragmentów na podstawie tych nazw i używa tych materiałów do renderowania. Można jednak zmieniać materiał używany w danym fragmencie danej encji za pomocą metod takich jak `SetCustomMaterial`.

Klasa `QuadEntity` reprezentuje umieszczony w przestrzeni 3D prostokąt zbudowany z dwóch trójkątów — tzw. *quad*. Jego powierzchnia może być rysowana dowolnym materiałem. Jego kształtem steruje szereg parametrów. Najważniejsza jest możliwość pracy jako tzw. *billboard* — prostokąt obrócony zawsze przodem do kamery. Mechanizm ten wykorzystywany był od dawna w grafice 3D do umieszczania w przestrzeni płaskich obrazów. Pierwsze gry trójwymiarowe (np. Doom) w ten sposób pokazywały postacie przeciwników. Obecnie jest to rzadziej spotykane, ale *billboard* może znaleźć wiele zastosowań. Parametry przechowywane przez klasę to:

- `DegreesOfFreedom` — liczba stopni swobody: 0 oznacza prostokąt zorientowany zawsze zgodnie z wektorami `RightDir` i `UpDir`, 1 oznacza automatyczne obracanie w kierunku kamery wokół osi pionowej ($Y$), 2 oznacza automatyczne obracanie w kierunku kamery wokół dwóch osi,
- `UseRealDir` — wartość logiczna określająca, czy jako kierunek do którego ma być automatycznie obracany prostokąt użyty ma zostać prawdziwy kierunek od środka prostokąta do kamery (`true`), czy kierunek patrzenia kamery niezależny od pozycji danej encji na ekranie (`false`),
- `RightDir` i `UpDir` to wektory znormalizowane wyznaczające kierunek „w prawo" i „do góry", według którego prostokąt ma być zorientowany,
- `Tex` to zakres koordynatów tekstury, która nakładana jest na prostokąt,
- `HalfSize` to połowa szerokości i wysokości prostokąta.

Subtelne różnice w sposobie automatycznego obracania się prostokąta w kierunku kamery zależnie od parametrów `DegreesOfFreedom` i `UseRealDir` najlepiej jest zilu-

strwać — pojazuje je zrzut ekranu ze specjalnie przygotowanej sceny rys. 4.6. Implementację algorytmu wyznaczającego orientację prostokąta zawiera funkcja `CalcBillboardDirections`.

Klasa `TextEntity` reprezentuje tekst umieszczony w przestrzeni 3D. Jej działanie podobne jest od klasy `QuadEntity`. Podobnie jak w niej, tekst również może być zorientowany na stałe w kierunku wyznaczonym przez wektory `RightDir` i `UpDir` bądź obracać się zawsze w kierunku kamery wg jednego lub dwóch stopni swobody. Ponadto dostępne są wszelkie możliwości formatowania tekstu dostępne również w module Gfx2D (automatyczne dzielenie na wiersze na granicy słowa, dosunięcie pionowe i poziome, podkreślenie, przekreślenie itd.), ponieważ klasa korzysta z funkcjonalności zasobu czcionki (patrz p. 2.4). Rozmiar czcionki (wysokość liter) jest określany w jednostkach globalnego układu współrzędnych.

`StripeEntity` to abstrakcyjna klasa bazowa, która reprezentuje wąski, podłużny pasek trójkątów w przestrzeni 3D ułożony wzdłuż określonej ściezki. Przykłady pokazuje rys. 4.7.

Zadaniem klasy pochodnej jest przechowywanie i zwracanie sekwencji punktów wyznaczających tą ścieżkę. Ponieważ „pasek", do efektownego przedstawienia na ekranie, musi posiadać grubość większą niż jeden piksel, specjalny algorytm układa trójkąty wzdłuż tej ścieżki tak, aby były one zorientowane przodem do kamery. Nie dla każdego przypadku możliwe jest wyznaczenie takich trójkątów, ale używany do tego algorytm, zawarty w metodzie `DrawFragmentGeometry`, dla większości rzeczywistych sytuacji działa dostatecznie dobrze tworząc niewiele widocznych zgięć.

Pasek jest rysowany z użyciem wybranego materiału. Koordynaty tekstury na jego powierzchni można ustawić, a także włączyć ich automatyczną animację w sposób płynny (co powoduje efekt przewijania) i/lub skokowy.

Klasy pochodne to:

- `LineStripeEntity` to złożona z prostych odcinków łamana łącząca podaną sekwencję punktów,
- `EllipseStripeEntity` to krzywa zamknięta reprezentująca elipsę, której orientację wyznaczają dwa wektory promieni,
- `UTermEntity` to otwarta lub zamknięta krzywa parametryczna opisana obiektem klasy `Uterm3`. *Uterm* (od ang. *Universal Term*) to funkcja parametru $t$ opisana wzorem:

$$v = A_2 \cdot t^2 + A_1 \cdot t + A_0 + B_2 \cdot \sin(B_1 \cdot t + B_0) \tag{2.1}$$

Funkcja ta, wyznaczana dla wartości skalarnej lub wektorowej, pozwala za pomocą 6 współczynników opisać bardzo szeroki wachlarz różnych kształtów, od linii prostej i paraboli, poprzez sinusoidę, aż po różnego rodzaju spirale w przestrzeni 3D. Klasy przechowujące te współczynniki i wyznaczające wg wydajnego

algorytmu wartość funkcji dla podanego parametru to Uterm, Uterm2, Uterm3 i Uterm4.

- Klasa CurveEntity reprezentuje krzywą w przestrzeni 3D wyznaczają przez sekwencję punktów. Dostępne są trzy rodzaje krzywych: kwadratowa (każdy segment wyznaczają 3 punkty), krzywa Beziera (każdy segment wyznaczają 4 punkty) oraz krzywa B–spline.

Klasa ParticleEntity reprezentuje efekt cząsteczkowy (ang. *Particle Effect*) [57]. Jest to efekt typu stanowego (patrz p. 1.1). Do rysowania cząstek używa alfa–blendingu i podanej tekstury. Podczas tworzenia obiektu trzeba podać liczbę cząsteczek. Włączenie funkcji LOD (od ang. *Level of Detail*) powoduje, że zależnie od odległości encji od kamery rysowany jest tylko pewien procent spośród wszystkich cząstek. Dokładne omówienie znaczenia parametrów struktury PARTICLE_DEF opisujących efekt oraz sposobu jego renderowania na GPU znajduje się w p. 3.5. Przykładowe efekty działania prezentuje zrzut ekranu 4.8.

**Klasy efektów postprocessingu** Kod zgromadzony w plikach Engine\Engine_pp.hpp i Engine\Engine_pp.cpp dostarcza klas implementujących efekty postprocessingu. W zaawansowanym silniku platforma do definiowania takich efektów powinna być uogólniona, tak aby można było łatwo dodawać nowe efekty wyprowadzając klasy pochodne od pewnej klasy bazowej efektu bądź nawet definiować je w całości za pomocą specjalnych plików tekstowych czy dedykowanego edytora. Efekty tego rodzaju są jednak bardzo różnorodne i każdy z nich wymaga innych czynności. Dlatego w celu uproszczenia implementacji, autor zdecydował się na osobne klasy realizujące poszczególne efekty postprocessingu, z których każda ma własny interfejs dostosowany do jego wymagań.

Klasy zdefiniowane przez moduł to:

- PpEffect — klasa bazowa efektów postprocessingu. Nie wykorzystuje ona polimorfizmu i nie definiuje jednolitego interfejsu, a jedynie dostarcza implementacji kilku procedur wspólnych dla niektórych efektów, jak wyrenderowanie prostokąta rozciągniętego na całej powierzchni ekranu (ang. *Fullscreen Quad*),
- PpColor — prosty efekt zamalowania całego ekranu w sposób półprzezroczysty na wybrany kolor,
- PpTexture — prosty efekt zamalowania całego ekranu w sposób półprzezroczysty wybraną teksturą,
- PpFunction — efekt przekształcenia całego obrazu za pomocą funkcji matematycznej,
- PpToneMapping — efekt *Tone Mapping*,
- PpBloom — efekt *Bloom*,
- PpFeedback — efekt sprzężenia zwrotnego,
- PpLensFlare — efekt błysku soczewek.

Sposób implementacji poszczególnych efektów postprocessingu opisuje rozdz. 3.6.

**Klasy dla przestrzeni otwartych** Ważną część silnika stanowi kod renderujący poszczególne elementy przestrzeni otwartych — teren, niebo, wodę, drzewa, trawę i opady atmosferyczne.

Kod zgromadzony w plikach `Engine\Fall.hpp` i `Engine\Fall.cpp` stanowi implementację efektu opadów atmosferycznych (np. deszcz, śnieg). Za jego realizację odpowiada klasa `Fall`. Utworzenie obiektu tej klasy wymaga podania wypełnionej struktury `FALL_EFFECT_DESC`, która opisuje parametry efektu. Sposób realizacji efektu opadów atmosferycznych opisuje rozdz. 3.8.

Kod zgromadzony w plikach `Engine\Grass.hpp` i `Engine\Grass.cpp` służy do renderowania trawy i innych niskich obiektów gęsto rozmieszczonych na powierzchni terenu (np. kwiaty, kamienie). Utworzenie obiektu klasy `Grass` wymaga podania nazwy pliku tekstowego w specjalnym formacie opisującym parametry efektu, a także nazwę pliku specjalnej tekstury gęstości (ang. *Density Map*) przedstawiającej rozmieszczenie trawy na terenie. Sposób realizacji trawy opisuje rozdz. 3.9.

Kod zgromadzony w plikach `Engine\Sky.hpp` i `Engine\Sky.cpp` służy do renderowania nieba. `BaseSky` to abstrakcyjna klasa bazowa służąca do renderowania nieba. Dziedzicząca po niej klasa `SolidSky` rysuje proste tło w jednolitym kolorze. Klasa `SkyboxSky` używa do rysowania tła sześciu tekstur mapując je na wewnętrznych powierzchniach ścian sześcianu otaczającego kamerę. Jest to tzw. sześcian nieba (ang. *Skybox*). Najbardziej złożona spośród podklas — `ComplexSky` — renderuje proceduralnie generowane niebo z gradientem na tle, chmurami oraz ciałami niebieskimi takimi jak Słońce czy Księżyc. Sposób implementacji tej klasy opisuje rozdz. 3.10.

Kod zgromadzony w plikach `Engine\Terrain.hpp` i `Engine\Terrain.cpp` służy do renderowania terenu. Utworzenie obiektu klasy `Terrain` wymaga podania szeregu parametrów. Pośród nich jest nazwa pliku specjalnej tekstury zawierającej mapę wysokości (ang. *Heightmap*), innej tekstury obrazującej rozmieszczenie poszczególnych form terenu (np. trawy, piasku, śniegu) oraz pliku tekstowego w specjalnym formacie opisującym te formy terenu. Na podstawie tych danych klasa generuje podczas inicjalizacji siatkę trójkątów tworzącą teren w przestrzeni 3D. Ponieważ jest to proces kosztowny obliczeniowo, wyliczona za pierwszym razem siatka jest zapisywana w osobnym pliku binarnym celem ponownego użycia. Sposób implementacji terenu opisuje rozdz. 3.7.

Kod zgromadzony w plikach `Engine\Trees.hpp` i `Engine\Trees.cpp` służy do renderowania drzew. Drzewa są generowane proceduralnie. Obiekt klasy `Tree` reprezentuje pojedynczy gatunek drzewa i umożliwia jego renderowanie. Utworzenie takiego obiektu wymaga podania wypełnionej struktury `TREE_DESC`, opisującej parametry kształtu i wyglądu drzewa. Sposób implementacji drzew opisuje rozdz. 3.11.

Kod zgromadzony w plikach `Engine\Water.hpp` i `Engine\Water.cpp` służy do ren-

derowania powierzchni wody. Klasa bazowa `WaterBase` przechowuje parametry opisujące jej wygląd i zawiera kod odpowiedzialny za rendering. Dziedzicząca z niej klasa `WaterEntity` jest encją wyświetlającą zorientowany w płaszczyźnie poziomowej prostokąt rysowany jako powierzchnia wody. To pozwala wstawić obszar wody w dowolne miejsce sceny, także do scen typu zamkniętego. Druga podklasa — `TerrainWater` — rysuje wodę jako część terenu. Sposób implementacji renderowania wody opisuje rozdz. 3.12.

Za renderowanie przestrzeni otwartych odpowiada klasa `TerrainRenderer` zdefiniowana w pliku `Engine\Engine.hpp`. Przechowuje ona obiekty klas: `Terrain`, `Grass`, `TerrainWater` oraz `Tree` i organizuje proces ich renderowania. Obiekt tej klasy jest z kolei przechowywany przez scenę.

**Klasy dla przestrzeni zamkniętych** Kod zgromadzony w plikach `Engine\QMap.hpp` i `Engine\QMap.cpp` to klasa `QMap`. Jest ona typem zasobu. Służy do wczytywania, przechowywania i udostępniania danych o mapie typu *Indoor* (przestrzeń zamknięta). Mapa to siatka trójkątów, podobnie jak model 3D. Różni się jednak od modeli zarówno pod względem znaczeniowym (przedstawia pomieszczenia i korytarze, których wnętrza przemierza użytkownik), jak i pod względem technicznym (stosuje podział przestrzeni, aby można było rysować tylko widoczną część, a nie całość geometrii). Informacje na temat formatu QMAP zawiera rozdz. 3.14.

Mapa może zostać utworzona w programie graficznym (np. Blender, 3ds Max, Maya) lub specjalnym edytorze map (np. QuArK, DeleD). Musi zostać zapisana w zaprojektowanym przez autora formacie pliku: QMAP. W tym celu napisana została wtyczka eksportująca siatkę z programu graficznego Blender do formatu tymczasowego QMAP.TMP, a program narzędziowy (p. 2.6) przetwarza taki plik do formatu docelowego QMAP.

Za renderowanie mapy, której dane udostępnia opisana wyżej klasa zasobu, odpowiada klasa `QMapRenderer` zdefiniowana w pliku `Engine\Engine.cpp`. Obiekt tej klasy jest przechowywany przez scenę, analogicznie do klasy `TerrainRenderer` w przypadku scen typu otwartego.

## 2.6. Narzędzia i eksportery

Komercyjne silniki gier, takie jak Unreal Engine, dostarczane są razem z zestawem programów narzędziowych i edytorów wspierających tworzenie gier. Autor opisanego tu silnika graficznego zdecydował się, dla uproszczenia, zrezygnować z tworzenia jakichkolwiek programów tego typu. Większość danych graficznych podaje się więc bądź w postaci plików tekstowych w specjalnie zaprojektowanych formatach, bądź w postaci tekstur wyznaczających za pomocą kolorów swoich pikseli np. mapę wysokości terenu, rozmieszczenie na terenie drzew czy trawy.

Pewne dane muszą być jednak dostarczone w postaci binarnej. Dotyczy to zwłaszcza siatek trójkątów przedstawiających modele. Ponieważ silnik wykorzystuje własny format plików modeli QMSH, potrzebna jest możliwość zapisywania danych w tym formacie. Tworzenie własny formatów modeli jest częstą praktyką i problem ten rozwiązywany jest zazwyczaj za pomocą eksportera, czyli wtyczki do programu graficznego 3D potrafiącej zapisać model od razu w formacie docelowym.

Autor wybrał inne rozwiązanie. Dla darmowego programu graficznego 3D — Blender — napisane zostały w języku Python wtyczki eksportujące siatkę do formatów pośrednich, nazwanych QMSH.TMP i QMAP.TMP. Są to pliki tekstowe przechowujące potrzebne dane w takiej postaci, w jakiej pobrane zostały wprost z Blendera. Dzięki temu wtyczki są bardzo proste. Ich kod znajduje się w katalogu `Plugins\Blender`.

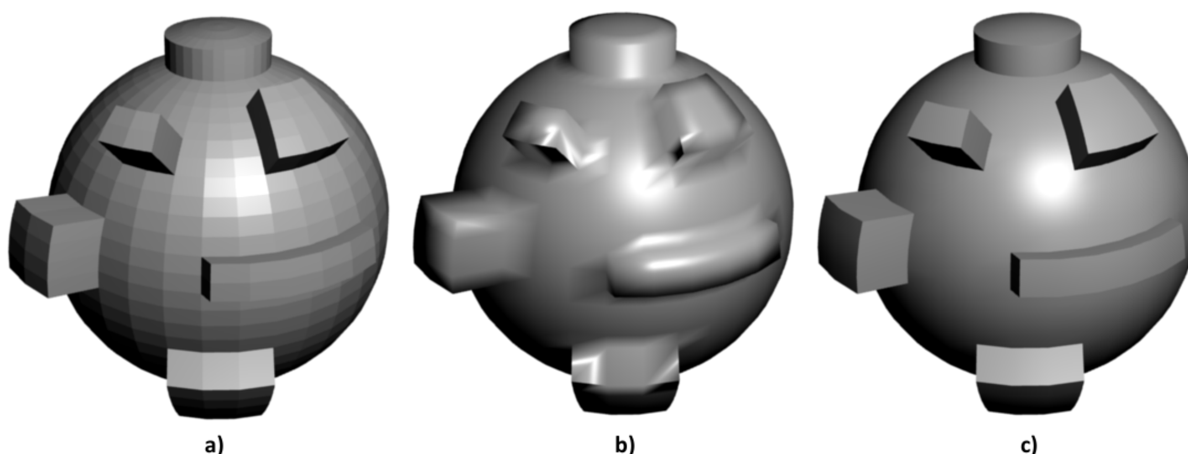
Dalsze przetwarzanie tekstowych plików w formatach pośrednich na pliki binarne w formatach docelowych odbywa się w osobnym programie konsolowym, napisanym już w C++. Nosi on nazwę **Tools**, a jego kod źródłowy zgromadzony jest w katalogu `src\Tools`. Sterowanie programem odbywa się za pomocą odpowiednich parametrów wiersza poleceń. Są trzy rodzaje dostępnych operacji: dotyczące siatek, map i tekstur. Pełna lista dopuszczalnych parametrów znajduje się w pliku `doc\Tools\Tools.txt`.

**Operacje na siatkach** Formatem pliku wejściowego siatki może być plik pośredni QMSH.TMP lub plik QMSH. Formatem wyjściowym jest zawsze QMSH. Siatka podczas wczytywania z formatu QMSH.TMP jest przetwarzana do formatu docelowego. W skład tego procesu przetwarzania wchodzą różnorodne obliczenia, w tym przekształcenie układu współrzędnych ze stosowanego przez Blender (prawoskrętny, $X$ w prawo, $Y$ w głąb, $Z$ do góry) do stosowanego przez silnik, standardowego dla DirectX (lewoskrętny, $X$ w prawo, $Y$ do góry, $Z$ w głąb).

Wyliczane są też wektory normalne, a na życzenie także styczne (*Tangent* i *Binormal*). Do obliczeń tych użyta została biblioteka C++ firmy NVIDIA — NvMeshMender. Bierze ona pod uwagę kąt między powierzchniami pozwalając na określenie granicy, poniżej której krawędź uznana zostaje za ostrą i wektory normalne nie są wygładzane. Jak ważna jest ta funkcja, obrazuje rys. 2.17.

Dodatkowo dostępne są parametry przekształcające siatkę, np. dokonujące translacji, rotacji, skalowania, centrowania na średniej lub medianie z pozycji wszystkich wierzchołków, przekształcania takiego aby siatka mieściła się wewnątrz podanego prostopadłościanu, skalowanie czasu animacji itd. Oprócz tego możliwe są proste operacje typu zmiana nazwy albo usunięcie podsiatki, animacji czy materiału.

Parametry `/I` oraz `/v` umożliwiają wypisanie na konsoli informacji o przetwarzanej siatce — liczby trójkątów, wierzchołków, animacji, kości, wielkości bryły otaczającej, listy podsiatek itd. Przykładowe wywołanie programu w celu przetworzenia siatki może wyglądać następująco:




Rys. 2.17. Wpływ sposobu obliczania wektorów normalnych na oświetlenie siatki. a) Brak wygładzania normalnych. b) Całkowite wygładzenie normalnych. c) Wygładzenie normalnych bazujące na granicznym kącie krawędzi.

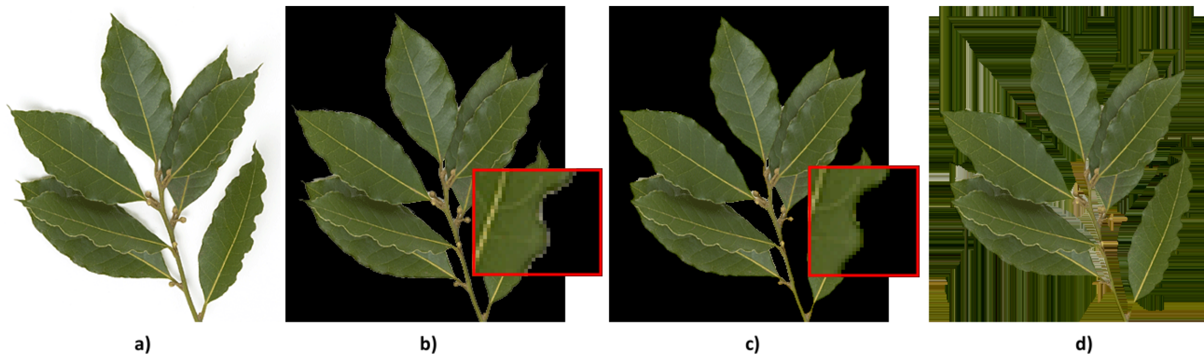
```
Tools.exe /Mesh /i=Monster01.qmsh.tmp
  /TransformToBox=-0.5,-0.5,-0.5;0.5,0.5;0.5
  "/ScaleTime=0.2|Animation:AnimacjaAtak"
  /o=Monster.qmsh /v /Tangents /MaxSmoothAngle=45
```

**Operacje na mapach** Przetwarzanie mapy z formatu pośredniego QMAP.TMP do formatu docelowego QMAP odbywa się w analogiczny sposób, jak przetwarzanie modeli. Służy do niego przełącznik `/Map`. Dodatkowo dostępne są opcje dostosowujące parametry drzewa ósemkowego stosowanego do podziału przestrzeni mapy, np. maksymalna głębokość drzewa.

**Operacje na teksturach** Podczas opracowania elementów graficznych wykorzystanych do zaprezentowania możliwości silnika wyniknęła potrzeba przetwarzania tekstur pewnymi algorytmami, których nie oferują zwyczajne programy graficzne. Dlatego opisany tu program konsolowy wzbogacony został o możliwość przetwarzania tekstur.

Jego możliwości w tym zakresie nie dublują funkcji dostępnych w zwyczajnych programach graficznych. Nie ma więc opcji do regulowania jasności, kontrastu czy nasycenia kolorów. Parametr `/Swizzle` pozwala na swobodną zamianę kanałów tekstury między sobą (czerwony, zielony, niebieski, alfa), ich kopiowanie oraz wypełnianie zerami bądź jedynkami. Parametr `/SharpenAlpha` służy do niwelowania niepożądanego efektu „otoczki" wokół krawędzi przezroczystego obiektu stosującego do renderowania test Alfa, często wykorzystywany przy przedstawianiu roślin. Użyty w tym celu wzór pochodzi z [70]. Parametr `/ClampTransparent` wypełnia kolory pikseli przezroczystych kolorem najbliższego nieprzezroczystego piksela w poziomie lub w pionie, co zapobiega „przeciekaniu" kolorów pikseli przezroczystych przy mipmappingu. Użycie dwóch ostatnich parametrów ilustruje rys. 2.18.

Polecania użyte do ich uzyskania to:




Rys. 2.18. Przetwarzanie tekstury z użyciem programu Tools. a) Oryginalne zdjęcie b) Tekstura z wyciętym obiektem, pokazana na tle koloru czarnego, ujawnia białą otoczkę na krawędziach. c) Zastosowanie parametru `/SharpenAlpha` niweluje niepożądany efekt. d) Zastosowanie parametru `/ClampTransparent` ostatecznie przygotowuje teksturę do zastosowania w renderowaniu rośliny.

```
Tools.exe /Texture /i=texture01.tga /SharpenAlpha=FFFFFF
  /o=texture02.tga /AlphaThreshold=192
Tools.exe /Texture /i=texture02.tga /ClampTransparent
  /o=texture03.tga /AlphaThreshold=192
```

# Rozdział 3

# Implementacja silnika

Ten rozdział przedstawia szczegóły implementacyjne wybranych efektów graficznych i innych ważniejszych elementów silnika. Nie sposób opisać dokładnie wszystkich algorytmów użytych w kodzie tej pracy.

## 3.1. Proces renderowania

**Algorytm renderowania** Właściwy proces renderowania sceny rozpoczyna się od metody `Scene::Draw` w pliku `Engine.cpp`. Jej głównym zadaniem jest jak najwydajniej zorganizować cały proces wplatając w odpowiednie miejsca wywołania służące do wykonania poszczególnych efektów postprocessingu. Na uwagę zasługuje decyzja podejmowana przez tą funkcję, czy potrzebne jest renderowanie sceny do tekstury, czy też można odrysować obiekty sceny wprost na buforze ramki (ang. *Back Buffer*). Renderowanie do tekstury pomocniczej jest wykonywane, jeśli włączony jest co najmniej jeden efekt postprocessingu, który tego wymaga. Wówczas obraz z tej tekstury zostaje w pewnym momencie przerysowany do bufora ramki za pomocą shadera PpShader (opisanego w p. 3.6), który wykonuje na pikselach obrazu wszystkie potrzebne operacje. Zarys omówionej metody w pseudokodzie przedstawia poniższy listing.

```
"function Scene::Draw"
Spytaj RunningOptimizer o ustawienia renderowania
Pobierz DRAW_DATA
if (postprocessing wyłączony):
  Rysuj scenę
else:
  if (efekty postprocessingu nie wymagają renderowania do tekstury):
    Rysuj scenę
    if (efekt błysku soczewek włączony):
      Zadaj zapytanie o widoczność słońca
  else:
    Pobierz teksturę celu renderowania
    Dla tekstury ustawionej jako cel renderowania:
      Rysuj scenę
      if (efekt błysku soczewek włączony):
        Zadaj zapytanie o widoczność słońca
```

```
      if (co najmniej jedna encja ciepła widoczna):
        Wyczyść kanał alfa celu renderowania
        Wyrenderuj encje ciepła do kanału alfa
    if (efekt Tone Mapping włączony):
      Oblicz jasność sceny
    if (efekt Bloom włączony):
      Przygotuj teksturę Bloom
    Przerysuj teksturę do bufora ramki za pomocą shadera PpShader
    if (efekt Bloom włączony):
      Narysuj efekt Bloom
    if (efekt sprzężenia zwrotnego włączony):
      Wykonaj efekt sprzężenia zwrotnego
    if (efekt błysku soczewek włączony):
      Wylicz intensywność błysku na podstawie widoczności słońca
      if (intensywność błysku > 0):
        Rysuj błysk soczewek
  if (efekt rysowania tekstury na ekranie włączony):
    Rysuj teksturę na ekranie
  if (efekt rysowania koloru na ekranie włączony):
    Rysuj kolor na ekranie
```

Struktura `DRAW_DATA` przechowuje listy elementów sceny przeznaczonych do wyrenderowania w danej klatce i służy do przekazywania tych danych między poszczególnymi wywołaniami rysującymi. Jej pola wypełniane początkowo to:

1. `std::vector<MaterialEntity*> MaterialEntities` — lista wskaźników na encje widoczne w zasięgu kamery, które używają do renderowania mechanizmu materiałów.
2. `std::vector<CustomEntity*> CustomEntities` — lista wskaźników na encje widoczne w zasięgu kamery rysowane w sposób niestandardowy.
3. `std::vector<HeatEntity*> HeatEntities` — lista wskaźników na encje ciepła widoczne w zasięgu kamery.
4. `std::vector<TREE_DRAW_DESC> Trees` — lista struktur opisujących widoczne w zasięgu kamery drzewa.
5. `QMapRenderer::FRAGMENT_DESC_VECTOR MapFragments` — lista struktur opisujących widoczne w zasięgu kamery fragmenty mapy.
6. `std::vector<uint> TerrainPatches` — lista indeksów do fragmentów terenu widocznych w zasięgu kamery.
7. `std::vector<SpotLight*> SpotLights` — lista świateł latarki, których zasięg działania koliduje z zasięgiem kamery.
8. `std::vector<PointLight*> PointLights` — lista świateł typu punktowego, których zasięg działania koliduje z zasięgiem kamery.

Na ich podstawie wypełniane są w dalszym etapie renderowania kolejne pola:

1. `std::vector<ENTITY_FRAGMENT> OpaqueEntityFragments` — lista struktur opisujących wszystkie fragmenty widocznych encji materiałowych renderowane materiałem typu nieprzezroczystego. Będą one rysowane w kolejności posortowanej wg materiału.

2. `std::vector<ENTITY_FRAGMENT> TranslucentEntityFragments` — lista struktur opisujących wszystkie fragmenty widocznych encji renderowanych materiałem typu półprzezroczystego i szkieletowego, wraz z encjami rysowanymi w sposób niestandardowy. Będą one rysowane w kolejności posortowanej wg odległości od kamery.

Wspomniana na powyższym listingu procedura pobierania struktury `DRAW_DATA` jest zaimplementowana w postaci metody `Scene::CreateDrawData`. Jej działanie w pseudokodzie ilustruje następujący listing:

```
"function Scene::CreateDrawData"
Wyczyść listy w DrawData
Entities = Octree.Wszystkie encje widoczne w zasięgu kamery
foreach (Encja in Entities):
  if (Encja is MaterialEntity):
    DrawData.MaterialEntities.Dodaj(Encja)
  else if (Encja is CustomEntity):
    DrawData.CustomEntities.Dodaj(Encja)
  else if (Encja is HeatEntity):
    DrawData.HeatEntities.Dodaj(Encja)
  else if (Encja is TreeEntity):
    DrawData.Trees.Dodaj(Struktura opisująca drzewo z Entity)
if (Mapa != null):
  DrawData.MapFragments = Mapa.Wszystkie fragmenty w zasięgu kamery
  DrawData.MapFragments.Sortuj_wg_materiału
if (Teren != null):
  DrawData.TerrainPatches = Teren.Indeksy widocznych fragmentów
  DrawData.Trees.Dodaj(Teran.Struktury opisujące drzewa w zasięgu
    kamery)
if (Oświetlenie włączone):
  foreach (Światło in Wszystkie światła punktowe na scenie):
    if (Światło.Aktywne i Światło.Kolor != CZARNY):
      if (Zasięg działania światła przecina zasięg widzenia kamery):
        DrawData.PointLights.Dodaj(Światło)
  foreach (Światło in Wszystkie światła Spot na scenie):
    if (Światło.Aktywne i Światło.Kolor != CZARNY):
      if (Zasięg działania światła przecina zasięg widzenia kamery):
        DrawData.SpotLights.Dodaj(Światło)
```

Wspomniana na pierwszym listingu procedura `Rysuj scenę` jest zaimplementowana w postaci metody `Scene::DrawAll`. Jej zadaniem jest odrysowanie na ustawionym aktualnie celu renderowania (czyli bezpośrednio na buforze ramki bądź na teksturze pomocniczej) wszystkich widocznych obiektów sceny, których listy otrzymuje w strukturze `DRAW_DATA`. Przebieg tego algorytmu ilustruje w pseudokodzie poniższy listing:

```
"function Scene::DrawAll"
if (Niebo != null)
  Niebo.Rysuj
Wyczyść Z-bufor
if (Mapa != null):
  foreach Fragment in DrawData.MapFragments:
    Mapa.Rysuj_fragment(Fragment, Przebieg bazowy)
if (Teren != null):
  foreach Fragment in DrawData.TerrainPatches:
    Teren.Rysuj_fragment(Fragment, Przebieg bazowy)
if (RunningOptimizer.Wykonywać_testy_zasłaniania_encji):
```

```
  Wykonaj testy zasłaniania encji
if (RunningOptimizer.Wykonywać_testy_zasłaniania_świateł):
  Wykonaj testy zasłaniania świateł
Spisz fragmenty encji materiałowych w DrawData
if (RunningOptimizer.Sortować_wg_materiału):
  DrawData.OpaqueEntityFragments.Sortuj_wg_materiału
foreach (Fragment in DrawData.OpaqueEntityFragments):
  Rysuj fragment, Przebieg bazowy
DrawData.Trees.Sortuj_wg_gatunku_drzewa
foreach (Drzewo in DrawData.Trees):
  Drzewo.Rysuj
if (Teren != null):
  Teren.Rysuj_trawę
if (Oświetlenie włączone):
  if (Światło_kierunkowe != null i Światło_kierunkowe.Aktywne i
    Światło_kierunkowe.Kolor != CZARNY):
    if (Shadow mapping włączony i Światło_kierunkowe.RzucaCień):
      Rysuj do shadow mapy światła kierunkowego
    if (Mapa != null i Mapa.Używa_oświetlenia):
      foreach (Fragment in DrawData.MapFragments):
        Mapa.RysujFragment(Fragment, Przebieg światła kierunkowego)
    if (Teren != null):
      foreach (Fragment in DrawData.TerrainPatches):
        Teren.RysujFragment(Fragment, Przebieg światła kierunkowego)
    foreach (Encja in DrawData.OpaqueEntityFragments):
      Encja.Rysuj(Przebieg światła kierunkowego)
  foreach (Światło in DrawData.PointLights):
    if (Światło.Aktywne i Światło.Kolor != CZARNY):
      if (Zasięg działania światła przecina zasięg widzenia kamery):
        ScissorRect = Wyznacz prostokąt zasięgu światła na ekranie
        if (Shadow mapping włączony i Światło.RzucaCień):
          Rysuj do shadow mapy światła punktowego
        Ustaw Scissor Test
        if (Mapa != null i Mapa.Używa_oświetlenia):
          foreach (Fragment in DrawData.MapFragments):
            if (Fragment.AABB przecina zasięg światła):
              Mapa.Rysuj_fragment(Fragment,
                Przebieg światła kierunkowego)
        if (Teren != null):
          foreach (Fragment in DrawData.TerrainPatches):
            if (Fragment.AABB przecina zasięg światła):
              Teren.Rysuj_fragment(Fragment,
                Przebieg światła kierunkowego)
        foreach (Fragment in DrawData.OpaqueEntityFragments):
          if (Fragment.Encja używa oświetlenia):
            if (Fragment.Encja.Sfera_otaczająca przecina
              zasięg światła):
              Rysuj fragment, Przebieg światła kierunkowego
        Wyłącz Scissor Test
  foreach (Światło in DrawData.SpotLights):
    if (Światło.Aktywne i Światło.Kolor != CZARNY):
      if (Zasięg działania światła przecina zasięg widzenia kamery):
        ScissorRect = Wyznacz prostokąt zasięgu światła na ekranie
        if (Shadow mapping włączony i Światło.RzucaCień):
          Rysuj do shadow mapy światła typu Spot
        Ustaw Scissor Test
        if (Mapa != null i Mapa.Używa_oświetlenia):
          foreach (Fragment in DrawData.MapFragments):
            if (Fragment.AABB przecina zasięg światła):
              Mapa.Rysuj_fragment(Fragment, Przebieg światła Spot)
        if (Teren != null):
          foreach (Fragment in DrawData.TerrainPatches):
            if (Fragment.AABB przecina zasięg światła):
              Teren.Rysuj_fragment(Fragment, Przebieg światła Spot)
```

```
        foreach (Fragment in DrawData.OpaqueEntityFragments):
          if (Fragment.Encja używa oświetlenia):
            if (Fragment.Encja.Sfera_otaczająca przecina zasięg
              światła):
              Rysuj fragment, Przebieg światła Spot
        Wyłącz Scissor Test
if (Mgła włączona):
  if (Mapa != null):
    foreach (Fragment in DrawData.MapFragments):
      Mapa.Rysuj_fragment(Fragment, Przebieg mgły)
  if (Teren != null):
    foreach (Fragment in DrawData.TerrainPatches):
      Teren.Rysuj_fragment(Fragment, Przebieg mgły)
  foreach (Fragment in DrawData.OpaqueEntityFragments):
    Rysuj fragment, Przebieg mgły
DrawData.TranslucentEntityFragments.Sortuj_wg_odległości_od_kamery
foreach (Fragment in DrawData.TranslucentEntityFragments):
  if (Fragment oznacza CustomEntity):
    Fragment.Rysuj jako CustomEntity
  else: // Fragment to fragment encji typu MaterialEntity
    if (Fragment.Materiał is WireframeMaterial):
      Fragment.Rysuj jako fragment encji z WireframeMaterial
    else // Fragment.Material is TranslucentMaterial
      Fragment.Rysuj jako fragment encji z TranslucentMaterial
if (Teren != null)
  Teren.Rysuj_wodę
if (Opady != null)
  Opady.Rysuj
```

Użyta w powyższym kodzie procedura „Rysuj do mapy cienia typu punktowego” jest zaimplementowana jako metoda `Scene::DoShadowMapping_Directional`. Jej zadaniem jest wyznaczenie zasięgu światła w kontekście aktualnej kamery, pobranie wszystkich obiektów sceny znajdujących się w tym zasięgu i ich wyrenderowanie do mapy cienia za pomocą przeznaczonego w tym celu przebiegu, a także skonstruowanie i zwrócenie macierzy używanych do późniejszego nałożenia mapy cienia podczas renderowania obiektów oświetlonym danym światłem. Procedura ta wygląda w pseudokodzie następująco:

```
"function Scene::DoShadowMapping_Directional"
Frustum = Zbuduj nowy frustum kamery uwzględniający zasięg światła
if (Mapa != null i Mapa.Rzuca_cień):
  MapFragments = Mapa.Fragmenty_widoczne(Frustum)
  if (RunningOptimizer.Sortować_wg_materiału):
    MapFragments.Sortuj_wg_materiału
Entities = Octree.Encje_rzucające_cień
foreach (Entity in Entities):
  if (Entity is MaterialEntity):
    foreach (Fragment in Entity.Fragments):
      if (Fragment.Materiał is OpaqueMaterial):
        EntityFragments.Dodaj(Fragment)
  else if (Entity is TreeEntity):
    Trees.Dodaj(Fragment)
if (Teren != null):
  Trees.Dodaj(Teran.Drzewa_rzucające_cień)
if (RunningOptimizer.Sortować_wg_materiału):
  EntityFragments.Sortuj_wg_materiału
Trees.Sortuj_wg_gatunku_drzewa
ViewProj = Oblicz macierz ViewProj światła kierunkowego
EngineServices.Pobierz mapę cienia
Dla mapy cienia ustawionej jako cel renderowania:
```

```
  foreach (Fragment in MapFragments):
    Rysuj fragment, Przebieg mapy cienia
  foreach (Fragment in EntityFragments):
    Rysuj fragment, Przebieg mapy cienia
  foreach (Tree in Trees):
    Rysuj Tree, Przebieg mapy cienia
m2 = Oblicz macierz tranformacji mapy cienia
Zwróć macierze ViewProj i m2
```

Metody wypełniające mapę cienia dla świateł typu punktowego i światła latarki wyglądają podobnie.

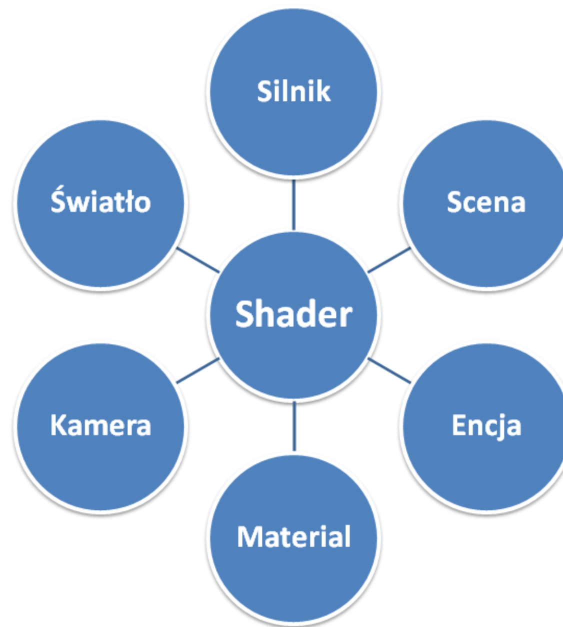
**Ustawienia renderowania** Rysowanie przez silnik każdej porcji geometrii składa się z dwóch etapów:

1. Ustawienie stanów renderowania Direct3D, w tym tekstur, shaderów i parametrów przekazanych do shaderów.
2. Ustawienie aktywnego bufora wierzchołków, bufora indeksów, formatu wierzchołka FVF i uruchomienie metody renderującej.

Ewentualna optymalizacja mogłaby polegać na ustawianiu tylko tych stanów urządzenia D3D, które zmieniły się od poprzedniego rysowania. Biblioteka Direct3D samodzielnie wykonuje jednak taką optymalizację, dlatego już samo pogrupowanie fragmentów przeznaczonych do narysowania wg zbliżonych parametrów (np. ich posortowanie wg używanego materiału) jest pewną optymalizacją. Lepszym rozwiązaniem byłoby podzielenie możliwych ustawień na grupy zależnie od częstości zmian (np. ustawienia zmieniane tylko raz w każdej klatce, ustawienia zmieniane tylko przy zmianie materiału, ustawienia zmieniane dla każdego rysowanego obiektu), ale autor nie zdecydował się na użycie tej techniki celem zapewnienia większej prostoty.

Pierwszy z wyżej wymienionych etapów wykonywany jest przez metodę `EngineServices::SetupState_Material` (poza przypadkami, kiedy encja sama odrysowuje siebie w niestandardowy sposób — dotyczy to klas pochodnych od `CustomEntity`). Otrzymuje ona wszystkie dane potrzebne do dokonania tych ustawień: rodzaj przebiegu, parametry renderowania sceny, obiekt kamery, obiekt materiału, parametry renderowania encji i obiekt światła. W tym miejscu muszą się spotkać parametry wszystkich obiektów biorących udział w renderowaniu (patrz rys. 3.1). Karta graficzna nie zna bowiem idei enkapsulacji. W danej chwili ustawiony może być tylko jeden aktywny Vertex Shader i Pixel Shader i musi on wykonywać wszystkie obliczenia, a przez parametry shaderów muszą być przekazane wszystkie dane potrzebne do jego działania. Proces ustawiania składa się z następujących etapów:

1. Sformułowany zostaje zestaw wartości makr preprocesora dla shadera.
2. Następuje pobranie shadera o określonych wartościach makr z obiektu shadera głównego klasy `MainShader`, który w razie potrzeby wczytuje go przy pierwszym użyciu z dysku bądź kompiluje z kodu źródłowego.




Rys. 3.1. Shader jako miejsce, w którym muszą spotkać się parametry wszystkich obiektów uczestniczących w procesie renderowania.

3. Ustawiane są parametry shadera.
4. Ustawiane są stany Direct3D.

**Rodzaje przebiegów** Proces renderowania sceny podzielić można na przebiegi (ang. *Pass*). Rodzaj przebiegu jest podstawowym makrem sterującym kompilacją shadera głównego. Shader główny zmienia prawie całą swoją funkcjonalność w zależności od typu przebiegu. Oto wykaz rodzajów przebiegów:

- `PASS_BASE = 0` — przebieg bazowy dla fragmentów encji rysowanych materiałem nieprzezroczystym. Wykonuje kilka czynności: wypełnia Z–bufor, rysuje powierzchnię oświetloną światłem rozproszonym (*Ambient*), rysuje teksturę emisji (*Emissive*) oraz teksturę mapowania środowiskowego (*Environmental Map*).
- `PASS_FOG = 1` — przebieg nakładający mgłę na fragmenty encji rysowanych materiałem nieprzezroczystym.
- `PASS_WIREFRAME = 2` — przebieg renderujący w całości fragmenty encji używające materiału szkieletowego `WireframeMaterial` (łącznie z mgłą).
- `PASS_TRANSLUCENT = 3` — przebieg renderujący w całości fragmenty encji używające materiału półprzezroczystego `TranslucentMaterial` (łącznie z mgłą).
- `PASS_DIRECTIONAL = 4` — przebieg dla fragmentów encji rysowanych materiałem nieprzezroczystym dodający oświetlenie od światła kierunkowego.
- `PASS_POINT = 5` — przebieg dla fragmentów encji rysowanych materiałem nieprzezroczystym dodający oświetlenie od światła punktowego.
- `PASS_SPOT = 6` — przebieg dla fragmentów encji rysowanych materiałem nieprzezroczystym dodający oświetlenie od światła latarki.
- `PASS_SHADOW_MAP = 7` — przebieg renderujący głębokość do mapy cienia.

- PASS_TERRAIN = 8 — przebieg renderujący teren. Rysuje powierzchnię oświetloną światłem rozproszonym (*Ambient*), światłem kierunkowym oraz mgłę.

Encje rysowane w sposób niestandardowy (CustomEntity), fragmenty encji materiałowych (MaterialEntity) używające materiałów szkieletowych (WireFrameMaterial), lub półprzezroczystych (TranslucentMaterial), a także drzewa i inne nietypowe elementy sceny są rysowane za jednym razem. Jedynie te fragmenty encji materiałowych, które używają materiału typu nieprzezroczystego (OpaqueMaterial), a także fragmenty terenu, podlegają oświetleniu i ich obraz jest budowany w wielu przebiegach.

W każdym takim przebiegu rysowana jest od nowa geometria danego fragmentu, z odpowiednim shaderem i odpowiednimi przekazanymi do niego parametrami. Przebieg bazowy odpowiada za narysowanie wszystkich elementów materiału niezależnych od oświetlenia. Następnie osobny przebieg dla każdego światła używa blendingu addytywnego, aby rozjaśnić piksele obiektu o oświetlenie danym światłem. Wreszcie, przebieg mgły dodaje zamglenie jego obrazu zależnie od odległości.

Wyróżnienie materiału półprzezroczystego jako osobny i niepodlegający oświetleniu było decyzją projektową podyktowaną uproszczeniem organizacji procesu renderowania sceny. Obiekty półprzezroczyste wymagają do prawidłowego przedstawienia, aby renderować je od razu w całości, w kolejności od najdalszych do najbliższych względem kamery, ponieważ nie współpracuje z nimi prawidłowo Z–bufor. Tymczasem proces renderowania oświetlonej geometrii jest w silniku zorganizowany wg kolejnych świateł, nie wg kolejnych obiektów. Zapewnia to większą wydajność oraz pozwala na przemian wypełniać i wykorzystywać mapę cienia. Gdyby obiekt półprzezroczysty miał podlegać oświetleniu wraz z rzucaniem na niego cienia, musiałyby do jego wyrenderowania istnieć i zostać wykorzystane jednocześnie mapy cienia ze wszystkich świateł.

Kolejność wykonywania poszczególnych przebiegów i innych wywołań renderujących można przedstawić w następujący sposób:

1. Niebo.
2. Przebieg bazowy (fragmenty mapy).
3. Fragmenty terenu.
4. Przebieg bazowy (fragmenty encji MaterialEntity z materiałem OpaqueMaterial).
5. Drzewa.
6. Trawa.
7. Dla kolejnych świateł: a) przebieg wypełniania mapy cienia, b) przebieg światła danego typu (fragmenty encji MaterialEntity z materiałem OpaqueMaterial, fragmenty mapy, fragmenty terenu).
8. Przebieg mgły (fragmenty encji MaterialEntity z materiałem OpaqueMaterial, fragmenty mapy, fragmenty terenu).

9. W kolejności od najdalszych względem kamery: Przebiegi materiałów szkieletowych i półprzezroczystych, encje rysowane we własnym zakresie.
10. Woda.
11. Opady atmosferyczne.

## 3.2. Implementacja shadera głównego

Kod źródłowy głównego shadera używanego do renderowania większości rodzajów elementów w silniku znajduje się w pliku `Engine\Multishaders\MainShader.fx`. Liczy 907 linii. Nie jest oczywiście kompilowany w całości. Znajduje się w nim wiele dyrektyw preprocesora dla kompilacji warunkowej `#if`, na podstawie których wybierane są fragmenty przeznaczone do kompilacji shadera z określonym zestawem makr. Jest to subtraktywna metoda budowania shaderów (p. 2.4). Wszystkie używane makra preprocesora, ich dopuszczalne wartości, znaczenie i zależności między nimi, jak również parametry przekazywane do shadera, ich znaczenie i zależności od makr są opisane w pliku `doc\Engine\MainShader.txt`.

Plik efektu `MainShader.fx` to kod w języku HLSL złożony z jednego Vertex Shadera i Pixel Shadera. Uproszczony potok renderowania Direct3D pokazujący przepływ danych między nimi znajduje się na rys. 3.2. Wierzchołki z wysłanej do renderowania siatki trójkątów trafiają wprost do Vertex Shadera. On ma możliwość dokonania obliczeń na danych każdego wierzchołka, a wyniki wysyła przez interpolatory do Pixel Shadera. Wyniki te są interpolowane między wierzchołkami na powierzchni trójkątów. Należą do nich m.in. pozycja i koordynaty tekstury. Tych ostatnich może być wiele, dlatego wszelkie niestandardowe dane, które trzeba przekazać do Pixel Shadera, są zapisywane jako dodatkowe koordynaty tekstury. Następnie zinterpolowane wartości trafiają do Pixel Shadera, który wykonywany jest dla każdego renderowanego piksela. Jego zadanie jest na podstawie otrzymanych danych wyliczyć finalny kolor piksela do wpisania do celu renderowania (ewentualnie z użyciem blendingu).




Rys. 3.2. Uproszczony model potoku renderowania Direct3D, pokazujący przepływ danych używanych przez Vertex Shader i Pixel Shader.

Strukturę wierzchołka stanowiącego wejście do Vertex Shadera pokazuje poniższy listing. Nie w każdej sytuacji przekazywane czy wykorzystywane są wszystkie z tych pól.

```
struct VS_INPUT
{
  float3 Pos : POSITION;
  half3 Normal : NORMAL;
  #if (TERRAIN == 1)
```

```
    half4 TerrainTextureWeights: COLOR0;
  #endif
  half BoneWeight0 : BLENDWEIGHT0;
  int4 BoneIndices : BLENDINDICES0;
  half2 Tex : TEXCOORD0;
  half3 Tangent : TEXCOORD1;
  half3 Binormal : TEXCOORD2;
};
```

- `Pos` to pozycja wierzchołka we współrzędnych lokalnych modelu.
- `Normal` to wektor normalny wierzchołka.
- `TerrainTextureWeights` to wagi dla blendingu 4 tekstur wykorzystywanych podczas renderowania terenu.
- `BoneWeight0` to waga wpływu pierwszej kości na wierzchołek. Waga wpływu drugiej kości wynosi `1−BoneWeight0`.
- `BoneIndices` zawiera indeksy dwóch kości wpływających na wierzchołek (wartość 0 oznacza brak kości).
- `Tex` to współrzędne tekstury.
- `Tangent` i `Binormal` to wektory styczne do powierzchni.

Struktura `VS_OUTPUT` przekazywana z Vertex Shadera do Pixel Shadera jest bardzo złożona. Jej pola są wybierane przez preprocesor zależnie od wartości makr. Jest to podyktowane kilkoma przesłankami. Liczba dostępnych interpolatorów jest ograniczona. Warto było nadać im nazwy zgodne z przeznaczeniem w danej sytuacji. Ponadto w strukturze tej nie powinno być pól nieużywanych, ponieważ kompilacja zakończy się błędem w przypadku niewypełnienia któregoś z pól, a ich wypełnianie dowolną wartością wymaga dodatkowych instrukcji Vertex Shadera.

Wyjściem Pixel Shadera jest pojedyncza wartość typu `half4` lub `float4` oznaczająca kolor (lub inne dane) przeznaczony do zapisania w aktualnym celu renderowania.

**Obliczenia oświetlenia** W przypadku przebiegów nr 4, 5, 6, oznaczających renderowanie geometrii oświetlonej światłem jednego z trzech rodzajów, do shadera przekazane zostają m.in. następujące makra:

- `PER_PIXEL` — wartość 1 mówi, że oświetlenie ma być wykonywane na poziomie pikseli, a nie wierzchołków.
- `NORMAL_TEXTURE` — wartość 1 mówi, że materiał posiada dodatkową teksturę z mapą normalnych. Ta funkcja działa tylko kiedy `PER_PIXEL==1`.
- `SPOT_SMOOTH` — wartość 1 mówi, że światło latarki ma zanikać płynnie od środka do krawędzi stożka.

oraz następujące parametry:

- `CameraPos` — pozycja kamery przekształcona do przestrzeni lokalnej obiektu.
- `DirToLight` — kierunek „do światła”, czyli zanegowany kierunek padania światła.

- LightPos — pozycja światła przekształcona do przestrzeni lokalnej obiektu.
- LightRangeSq — kwadrat zasięgu działania światła.
- LightCosFov2 — cosinus połowy kąta rozchylenia snopa światła latarki.
- LightCosFov2Factor — współczynnik 1 / (1 - LightCosFov2).
- NormalTexture — tekstura z mapą normalnych.

Na ich podstawie wyliczane są wartości potrzebne do wykonania cieniowania. Na poniższych listingach MyDirToLight oznacza znormalizowany wektor kierunku od danego wierzchołka do światła w przestrzeni modelu, a Attn to współczynnik $[0;1]$ odpowiedzialny za zanikanie światła wraz z odległością, a w przypadku światła latarki dodatkowo wraz z kątem. Dla oświetlenia typu *per vertex* obliczenia wykonuje Vertex Shader:

```
half3 MyDirToLight;
half Attn;
#if (PASS == 4) // Światło Directional
  MyDirToLight = DirToLight;
  Attn = 1;
#elif (PASS == 5 || PASS == 6) // Światło Point, Spot
  float3 VecToLight = (float3)LightPos - InputPos;
  MyDirToLight = normalize(VecToLight);
  Attn = 1 - min(1, dot(VecToLight, VecToLight) / LightRangeSq);
  #if (PASS == 6) // Światło Spot
    half LightDirDot = max(0, dot(DirToLight, MyDirToLight));
    #if (SPOT_SMOOTH == 0)
      Attn *= step(LightCosFov2, LightDirDot);
    #else
      Attn *= max(0, (LightDirDot - LightCosFov2) *
        LightCosFov2Factor);
    #endif
  #endif
#endif
```

W przypadku oświetlenia typu *per pixel*, Vertex Shader korzysta z otrzymanych w wierzchołku wektorów normalnego i stycznych, aby utworzyć macierz zdolną przekształcać wektory z przestrzeni modelu (ang. *Object Space* lub *Model Space*) do przestrzeni stycznej (ang. *Tangent Space*). Następnie używa jej, aby przekształcić do przestrzeni stycznej wszystkie wektory biorące udział w oświetleniu i w takiej postaci przekazuje je przez interpolatory do Pixel Shadera. Na tym kończy się jego zadanie. Podkreślić należy, że przez interpolator przechodzą wektory wraz z długością i dopiero w Pixel Shaderze są normalizowane dla otrzymania samego kierunku. Tylko wtedy oświetlenie liczone jest prawidłowo.

```
float3x3 TangentSpace;
TangentSpace[0] = InputTangent;
TangentSpace[1] = InputBinormal;
TangentSpace[2] = InputNormal;
#if (PASS == 4) // Światło Directional
  Out.DirToLight = mul(TangentSpace, DirToLight);
#endif
#if (PASS == 5 || PASS == 6) // Światło Point, Spot
  float3 VecToLight = LightPos - InputPos;
  Out.VecToLight = mul(TangentSpace, VecToLight);
```

```
  #if (PASS == 6) // Światło Spot
    Out.VecToLight_Model = VecToLight;
  #endif
#endif
#if (SPECULAR != 0)
  half3 VecToCam = (float3)CameraPos - InputPos;
  Out.VecToCam = mul(TangentSpace, VecToCam);
#endif
```

Dalej, w przypadku oświetlenia *per pixel*, Pixel Shader odbiera zinterpolowane wektory i wykorzystuje je do obliczeń oświetlenia podobnych do przedstawionych wyżej, wykonywanych przez Vertex Shader dla oświetlenia *per vertex*. Ponadto, sampluje na początku teksturę mapy normalnych (jeśli jest obecna), aby otrzymać wektor normalny danego miejsca powierzchni [22]. Wektor ten jest wyrażony w przestrzeni stycznej, tak jak wektory otrzymane od Vertex Shadera.

```
half3 Normal;
#if (NORMAL_TEXTURE == 1)
  Normal = tex2D(NormalSampler, In.Tex)*2-1;
#else
  Normal = half3(0, 0, 1);
#endif
Normal = normalize(Normal);

half3 MyDirToLight;
half Attn;
#if (PASS == 4) // Światło Directional
  MyDirToLight = In.DirToLight;
  Attn = 1;
#elif (PASS == 5 || PASS == 6) // Światło Point, Spot
  half3 VecToLight = In.VecToLight;
  MyDirToLight = normalize(VecToLight);
  Attn = 1 - saturate(dot(VecToLight, VecToLight) / LightRangeSq);
  #if (PASS == 6) // Światło Spot
    half LightDirDot = dot(
      DirToLight, normalize(In.VecToLight_Model));
    #if (SPOT_SMOOTH == 0)
      Attn *= step(LightCosFov2, LightDirDot);
    #else
      Attn *= saturate((LightDirDot - LightCosFov2) *
        LightCosFov2Factor);
    #endif
  #endif
#endif
```

**Obliczenia cieniowania** Dane wyliczone w kodzie z poprzedniego podrozdziału służą dalej do obliczeń cieniowania, czyli wpływu światła na powierzchnię z uwzględnieniem parametrów materiału. Potrzebne do tego makra to:

- `HALF_LAMBERT` — wartość 1 mówi, że do oświetlenia ma zostać użyty wzór *Half–Lambert* [18].
- `SPECULAR` — wybór rodzaju odblasku. 0 oznacza brak, 1 oznacza zwykły odblask *Specular*, 2 i 3 to oświetlenie anizotropowe (ang. *Anisotropic Lighting*).
- `GLOSS_MAP` — wartość 1 mówi, że kanał alfa podstawowej tekstury ma oznaczać intensywność oblasku. Ta funkcja działa tylko, kiedy `SPECULAR!=0`.

- DIFFUSE_TEXTURE — wartość 1 oznacza, że materiał posiada podstawową teksturę koloru. Jeśli jej nie posiada, rysowany jest jednolitym kolorem.
- COLOR_MODE — tryb modyfikacji koloru podstawowego przez *TeamColor* encji.

Parametry uczestniczące w obliczeniach cieniowania to:

- DiffuseTexture — podstawowa tekstura koloru.
- DiffuseColor — jednolity kolor materiału, używany kiedy nie ma tekstury.
- TeamColor — kolor zapamiętany dla danej encji.
- LightColor — kolor światła.
- SpecularColor — kolor odblasku danego materiału.
- SpecularPower — wykładnik odblasku.

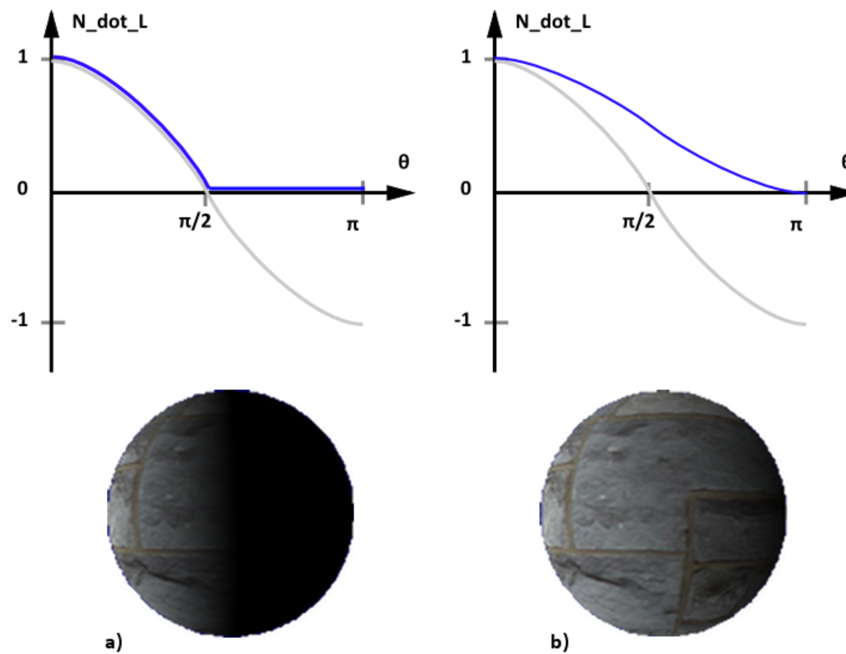
Funkcja pomocnicza ProcessTeamColor przetwarza kolor podstawowy materiału, odczytany wcześniej z tekstury lub stałej i przekazany jako parametr, uwzględniając TeamColor encji i sposób jego wpływu na wygląd powierzchni.

```
half4 ProcessTeamColor(half4 DiffuseColor)
{
  #if (COLOR_MODE == 1)
    half4 R = DiffuseColor * TeamColor;
    R.a = DiffuseColor.a;
    return R;
  #elif (COLOR_MODE == 2)
    half4 R = lerp(DiffuseColor, TeamColor, DiffuseColor.a);
    R.a = DiffuseColor.a;
    return R;
  #else // COLOR_MODE == 0
    return DiffuseColor;
  #endif
}
```

W przypadku oświetlenia *per vertex*, obliczeń cieniowania dokonuje Vertex Shader.

```
half N_dot_L = dot(InputNormal, MyDirToLight);
#if (HALF_LAMBERT == 1)
  N_dot_L = N_dot_L * 0.5 + 0.5;
#endif
half Diffuse = N_dot_L * Attn;
Out.Diffuse = Diffuse * LightColor;
#if (SPECULAR != 0)
  half3 MyDirToCam = normalize(CameraPos - InputPos);
  half3 Half = normalize(MyDirToLight + MyDirToCam);
  half N_dot_H = dot(InputNormal, Half);
  half Specular = pow(N_dot_H, SpecularPower);
  Out.Specular = Specular * LightColor * Diffuse * SpecularColor;
#endif
```

Dwie wartości wyliczane przez ten kod to kolor oświetlenia rozproszonego Diffuse i kolor odblasku Specular. Ten pierwszy powstaje zgodnie z prawem Lamberta. Mówi ono, że intensywność światła rozproszonego padającego na powierzchnię jest wprost proporcjonalne do cosinusa kąta $\theta$ między kierunkiem do źródła światła, a wektorem normalnym do powierzchni w danym miejscu. Ten kąt z kolei jest równy iloczynowi skalarnemu tych dwóch wektorów. W powyższym kodzie odpowiada mu zmienna




Rys. 3.3. Oświetlenie *Diffuse*: a) standardowe, b) *Half-Lambert*.

`N_dot_L`. Dodatkowa modyfikacja oświetlenia to *Half–Lambert* [18] pokazany na rys. 3.3.

Druga wartość to odblask `Specular`, wyliczany metodą Blinna [15]. W oryginalnej metodzie Phonga [14] brany jest kąt między wektorem światła odbitym od powierzchni a wektorem do kamery, podniesiony do potęgi. Metoda Blinna wymaga mniej obliczeń przybliżając ten wynik poprzez podniesienie do potęgi kąta między wektorem normalnym a tzw. wektorem połówkowym `Half`, który powstaje przez uśrednienie wektora do światła i wektora do kamery.

Pixel Shader sampluje jedynie teksturę podstawową koloru i łączy ją z tymi dwoma kolorami oświetlenia wyliczonymi przez Vertex Shader, zwracając wynik końcowy.

```
Out = DiffuseTextureColor * In.Diffuse;
#if (SPECULAR != 0)
  #if (GLOSS_MAP == 1)
    Out += In.Specular * DiffuseTextureColor.a;
  #else
    Out += In.Specular;
  #endif
#endif
```

W przypadku oświetlenia *per pixel*, podobne obliczenia wykonuje w całości Pixel Shader. Są jednak między nimi dwie istotne różnice. Po pierwsze, wszystkie obliczenia są wykonywane w przestrzeni stycznej, a nie w przestrzeni modelu. Po drugie, wektor normalny pobierany jest z tekstury mapy normalnych, a nie wprost z wierzchołka.

```
half N_dot_L = dot(Normal, MyDirToLight);
#if (HALF_LAMBERT == 1)
  N_dot_L = N_dot_L * 0.5 + 0.5;
#endif
half Diffuse = N_dot_L * Attn;
#if (NORMAL_TEXTURE == 1 && HALF_LAMBERT == 0)
```

```
  Diffuse *= (dot(half3(0,0,1), MyDirToLight) >= 0);
#endif
half4 OutDiffuse = Diffuse * LightColor;

half4 OutSpecular;
#if (SPECULAR != 0)
  float3 DirToCam = normalize(In.VecToCam);
  half N_dot_H;
  #if (SPECULAR == 1) // Zwykły odblask
    half3 Half = normalize(MyDirToLight + DirToCam);
    N_dot_H = dot(Normal, Half);
  #elif (SPECULAR == 2) // Anisotropic Lighting po Tangent
    half LdotT = MyDirToLight.x;
    half VdotT = DirToCam.x;
    N_dot_H = sqrt(1 - LdotT*LdotT) * sqrt(1 - VdotT*VdotT) -
      LdotT*VdotT;
  #else // (SPECULAR == 3) // Anisotropic Lighting po Binormal
    half LdotT = MyDirToLight.y;
    half VdotT = DirToCam.y;
    N_dot_H = sqrt(1 - LdotT*LdotT) * sqrt(1 - VdotT*VdotT) -
      LdotT*VdotT;
  #endif
  half Specular = pow(N_dot_H, SpecularPower);
  OutSpecular = Specular * LightColor * Diffuse * SpecularColor;
#endif

Out = DiffuseTextureColor * OutDiffuse;
#if (SPECULAR != 0)
  #if (GLOSS_MAP == 1)
    Out += OutSpecular * DiffuseTextureColor.a;
  #else
    Out += OutSpecular;
  #endif
#endif
```

Na uwagę w powyższym kodzie zasługują obliczenia oświetlenia anizotropowego. Może ono działać tylko kiedy oświetlenie jest *per pixel*, ale nie ma mapy normalnych. Współczynnik odblasku jest wtedy wyliczany w inny sposób niż wg opisanego wyżej wzoru Blinna i aproksymuje oświetlenie materiału o mikrostrukturze złożonej z włókien (np. tkanina, włosy, polerowany metal) ułożonych wzdłuż wektorów stycznych siatki — do wyboru *Tangent* lub *Binormal*. Wzory te zostały wyprowadzone na podstawie [19, 20].

**Obliczenia cienia** Rzucanie cienia zostało w silniku zrealizowane za pomocą techniki mapy cienia, znanej jako *Shadow Mapping*. Do mechanizmu cieni shader główny używa następujących makr:

- `SHADOW_MAP_MODE` to sposób zapisywania wartości do mapy cienia (p. niżej).
- `VARIABLE_SHADOW_FACTOR` równy 1 oznacza, że stopień przyciemnienia geometrii w cieniu nie jest stały, ale maleje wraz z odległością.
- `POINT_SHADOW_MAP` równy 1 oznacza, że rysowany jest cień dla światła punktowego, w którym wykorzystywana jest sześcienna tekstura mapy cienia.

oraz następujących parametrów:

- `ShadowFactor` to współczynnik jasności geometrii znajdującej się w cieniu.

- `ShadowFactorA`, `ShadowFactorB` to współczynniki funkcji liniowej opisującej zanikanie powyższego współczynnika jasności wraz z odległością od kamery.
- `ShadowMapTexture` to tekstura mapy cienia.
- `ShadowMapMatrix` to macierz przekształcająca punkty do współrzędnych mapy cienia.
- `ShadowMapSize` i `ShadowMapSizeRcp` to rozmiar i odwrotność rozmiaru mapy cienia, w pikselach.
- `ShadowEpsilon` to wartość tolerancji zapobiegająca artefaktom wynikającym ze skończonej dokładności i rozdzielczości mapy cienia.
- `LightRangeSq_World` to zasięg światła punktowego w przestrzeni świata.

Technika mapy cieni składa się z dwóch etapów. W pierwszym etapie osobny przebieg renderuje całą geometrię do tekstury mapy cienia, gdzie jako kolory pikseli zakodowane zostają odległości/głębokości od źródła światła do najbliższej geometrii.

W przypadku świateł punktowych używana jest tekstura sześcienna i zapisywana jest w niej odległość od pozycji światła do punktu. W przypadku pozostałych typów świateł dokonywane jest przekształcenie wraz z rzutowaniem perspektywicznym, które sprowadza geometrię do przestrzeni światła (ang. *Light Space*) i zapisywana jest głębokość, tak jak w Z–buforze. Ponieważ ze względu na sposób działania interpolacji danych przekazywanych od Vertex Shadera do Pixel Shadera nie można przekazać tam wprost odległości ani głębokości, przekazywane są inne dane. W przypadku światła punktowego jest to wektor od punktu do światła, z którego Pixel Shader wylicza długość. W przypadku innych typów świateł są to komponenty $(z, w)$ pozycji wierzchołków po transformacji we współrzędnych jednorodnych, a Pixel Shader dokonuje dzielenia $z/w$.

Oto fragment kodu Vertex Shadera używany w przebiegu renderowania do mapy cienia:

```
#if (POINT_SHADOW_MAP == 1)
  Out.VecToLight = InputPos - LightPos;
#else
  Out.ShadowMapZW = Out.Pos.zw;
#endif

#if (ALPHA_TESTING == 1)
  #if (TEXTURE_ANIMATION == 1)
    Out.Tex = mul(float4(In.Tex, 0, 1), (float4x2)TextureMatrix);
  #else
    Out.Tex = In.Tex;
  #endif
#endif
```

Na uwagę zasługuje fakt, że choć w omawianym przebiegu nie są potrzebne kolory, koordynaty tekstury muszą zostać przekazane, a tekstura podstawowa samplowana w Pixel Shaderze w przypadku kiedy materiał stosuje test Alfa. Wówczas bowiem kanał Alfa tekstury podstawowej wyznacza miejsca przezroczyste.

Oto kod Pixel Shadera używany w przebiegu renderowania do mapy cienia:

```
#if (POINT_SHADOW_MAP == 1)
  float v = dot(In.VecToLight, In.VecToLight) / LightRangeSq;
#else
  float v = In.ShadowMapZW.x / In.ShadowMapZW.y; // z/w
#endif
#if (SHADOW_MAP_MODE == 1) // Wykorzystaj składową R
  Out = float4(v, v, v, v);
#elif (SHADOW_MAP_MODE == 2) // Spakuj do składowych RGB
  Out = v * float4( 256*256, 256, 1, 0 );
  Out = frac(Out);
#endif

#if (ALPHA_TESTING == 1 && SHADOW_MAP_MODE == 1)
  float4 DiffuseTextureColor;
  #if (DIFFUSE_TEXTURE == 1)
    DiffuseTextureColor = tex2D(DiffuseSampler, In.Tex);
  #else
    DiffuseTextureColor = DiffuseColor;
  #endif
  Out.a = DiffuseTextureColor.a;
#endif
```

Omówienie techniki mapy cieni nie leży w zakresie tej pracy. Na uwagę zasługuje natomiast sposób zapisywania wartości odległości/głębokości do mapy cienia. Wartość ta, podzielona przez maksimum dla sprowadzenia do zakresu $0 \ldots 1$, musi zostać wpisana do tekstury jak kolor piksela. Jej wpisanie do jednej ze składowych ARGB zwyczajnej tekstury nie wchodzi jednak w grę, ponieważ precyzja 8 bitów na komponent jest niewystarczająca. Najlepszym wyjściem jest w takiej sytuacji użycie tekstury w formacie o zwiększonej precyzji i liczbie kanałów ograniczonej do 1 lub 2, takich jak `D3DFMT_R32F` albo `D3DFMT_G16R16`. Konfiguracji takiej odpowiada wartość makra `SHADOW_MAP_MODE=1`.

Niektóre z takich formatów są niestety niedostępne do wykorzystania jako cel renderowania na niektórych modelach kart graficznych, na których silnik powinien z założenia działać prawidłowo (np. GeForce FX 5200). Dlatego wprowadzone zostało dodatkowe rozwiązanie, bardziej kosztowe obliczeniowo ale bardziej kompatybilne, któremu odpowiada wartość makra `SHADOW_MAP_MODE=2`. Polega ono na spakowaniu wartości odległości/głębokości do 3 składowych tekstury (RGB) i pozwala na wykorzystanie standardowej tekstury w formacie `D3DFMT_A8R8G8B8`.

Drugi etap polega na wykorzystaniu tekstury mapy cienia podczas renderowania geometrii oświetlonej danym światłem do stwierdzenia, czy dane miejsce jest zasłonięte (jest w cieniu). Czynność tą wykonuje funkcja `SampleShadowMap`:

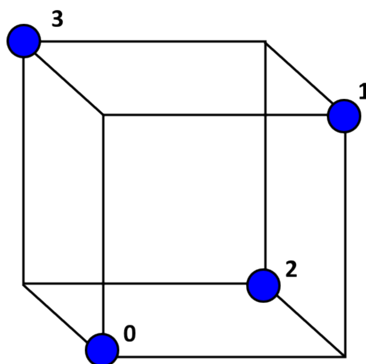
```
half SampleShadowMap(VS_OUTPUT In)
{
  const float4 RGBA_Factors = float4(1/256/256, 1/256, 1, 0);
  #if (PASS == 5) // Światło Point - Cube Map
    float3 VecFromLight = In.VecFromLight_World;
    float DistFromLight = dot(VecFromLight, VecFromLight);
    float3 ShadowTexC = VecFromLight;
    float4 samples;
    ShadowTexC = normalize(ShadowTexC);
    float3 lerps = frac(ShadowTexC * ShadowMapSize);
    #if (SHADOW_MAP_MODE == 1) // Składowa R
```

```
      samples.x = texCUBE(ShadowMapSampler, ShadowTexC +
        float3(0, 0, 0));
      samples.y = texCUBE(ShadowMapSampler, ShadowTexC +
        float3(ShadowMapSizeRcp, ShadowMapSizeRcp, 0));
      samples.z = texCUBE(ShadowMapSampler, ShadowTexC +
        float3(ShadowMapSizeRcp, 0, ShadowMapSizeRcp));
      samples.w = texCUBE(ShadowMapSampler, ShadowTexC +
        float3(0, ShadowMapSizeRcp, ShadowMapSizeRcp));
    #elif (SHADOW_MAP_MODE == 2) // Składowe RGB
      samples.x = dot(RGBA_Factors, texCUBE(ShadowMapSampler,
        ShadowTexC + float3(0, 0, 0)));
      samples.y = dot(RGBA_Factors, texCUBE(ShadowMapSampler,
        ShadowTexC + float3(ShadowMapSizeRcp, ShadowMapSizeRcp, 0)));
      samples.z = dot(RGBA_Factors, texCUBE(ShadowMapSampler,
        ShadowTexC + float3(ShadowMapSizeRcp, 0, ShadowMapSizeRcp)));
      samples.w = dot(RGBA_Factors, texCUBE(ShadowMapSampler,
        ShadowTexC + float3(0, ShadowMapSizeRcp, ShadowMapSizeRcp)));
    #endif
    half4 compared_samples =
      (DistFromLight / LightRangeSq_World) < (samples + ShadowEpsilon);
    return (compared_samples.x + compared_samples.y +
      compared_samples.z + compared_samples.w) / 4;
  #else // Światło inne - zwykła tekstura
    float4 ShadowMapPos = saturate(In.ShadowMapPos / In.ShadowMapPos.w);
    float2 texelpos = ShadowMapPos.xy * ShadowMapSize;
    float2 lerps = frac(texelpos);
    float4 samples;
    #if (SHADOW_MAP_MODE == 1) // Składowa R
      samples.x = tex2D(ShadowMapSampler, ShadowMapPos.xy).r;
      samples.y = tex2D(ShadowMapSampler, ShadowMapPos.xy +
        half2(ShadowMapSizeRcp, 0              )).r;
      samples.z = tex2D(ShadowMapSampler, ShadowMapPos.xy +
        half2(0,                ShadowMapSizeRcp)).r;
      samples.w = tex2D(ShadowMapSampler, ShadowMapPos.xy +
        half2(ShadowMapSizeRcp, ShadowMapSizeRcp)).r;
    #elif (SHADOW_MAP_MODE == 2) // Składowe RGB
      samples.x = dot(RGBA_Factors, tex2D(ShadowMapSampler,
        ShadowMapPos.xy));
      samples.y = dot(RGBA_Factors, tex2D(ShadowMapSampler,
        ShadowMapPos.xy + half2(ShadowMapSizeRcp, 0              )));
      samples.z = dot(RGBA_Factors, tex2D(ShadowMapSampler,
        ShadowMapPos.xy + half2(0,                ShadowMapSizeRcp)));
      samples.w = dot(RGBA_Factors, tex2D(ShadowMapSampler,
        ShadowMapPos.xy + half2(ShadowMapSizeRcp, ShadowMapSizeRcp)));
    #endif
    float4 compared_samples = (ShadowMapPos.zzzz <= samples);
    return lerp(
      lerp(compared_samples.x, compared_samples.y, lerps.x),
      lerp(compared_samples.z, compared_samples.w, lerps.x),
      lerps.y).r;
  #endif
}
```

Powyższa funkcja wykonuje filtrowanie próbek pobranych z mapy cienia. Nie można rozwiązać tego za pomocą standardowego filtrowania tekstur, jakie oferuje sprzęt graficzny, ponieważ interpolacja wartości zapisanych w mapie cienia nie ma sensu. Interpolacja musi nastąpić po wykonaniu porównania pobranej wartości z wartością odniesienia. Autor użył tutaj, ze względu na wydajność, najprostszego sposobu wygładzania cieni — PCF (ang. *Percentage Closer Filtering*) dla 4 próbek [30]. Wsparcie sprzętowe dla takiego PCF istnieje, ale jest obecne tylko na kartach graficznych firmy

NVIDIA jako niestandardowe rozszerzenie i dlatego nie zostało wykorzystane [31].

Na uwagę zasługuje trudność w filtrowaniu sześciennej tekstury mapy cienia, wykorzystywanej dla świateł punktowych. Użyty został w tym celu autorski pomysł na modyfikowanie wektora 3D używanego do adresowania takiej tekstury w sposób przypominający wybór 4 spośród 8 wierzchołków sześcianu o wielkości 1 teksela, jak to pokazuje rys. 3.4. Otrzymany w ten sposób efekt nie jest bardzo dobry, ale jest lepszy niż całkowity brak filtrowania, kiedy to teksele mapy cienia są widoczne jako ostre „schodki” wyznaczające krawędź cienia.




Rys. 3.4. Sposób adresowania tekstury sześciennej mapy cienia podczas próbkowania dla PCF.

Wzmianka należy się też problemowi przezroczystości. Obiekty półprzezroczyste nie podlegają w opisanym silniku oświetleniu, a więc także nie rzucają cienia. Obiekty wykorzystujące test Alfa mogą jednak to robić, co jest bardzo istotne podczas rysowania drzew, trawy i innych obiektów tego rodzaju. Tymczasem, zależnie od karty graficznej, test alfa nie zawsze działa w przypadku tekstur o formatach takich jak `D3DFMT_R32F`. Rozwiązanie tego problemu mogłoby polegać na „ręcznym” wykonaniu tego testu poprzez anulowanie rysowania pikseli w Pixel Shaderze instrukcją `texkill`, której odpowiada funkcja HLSL `clip`.

**Obliczenia mgły** W grafice 3D czasu rzeczywistego najczęściej stosowana jest mgła, której intensywność wzrasta wraz z odległością/głębokością liniowo lub wykładniczo. W opisywanym silniku zaimplementowana jest tylko mgła liniowa. Jej jedynym parametrem, oprócz koloru, jest procent odległości, w której się zaczyna. Kończy się zawsze na końcu zasięgu kamery (*Z-Far*).

Parametry dla shadera głównego biorące udział w renderowaniu mgły to:

- `FogColor` — kolor mgły,
- `FogFactors` — wyliczone przez CPU współczynniki równania liniowego do intensywności mgły w funkcji $z = [0; 1]$.

W przypadku renderowania z użyciem materiału półprzezroczystego, kiedy końcowy obraz powstaje w pojedynczym przebiegu, obliczenia mgły są wplecione w pozostałe obliczenia cieniowania. Jeśli natomiast obiekt rysowany jest materiałem nieprze-

zroczystym, po narysowaniu jego obrazu będącego sumą przebiegów dla poszczególnych świateł następuje osobny przebieg mgły (PASS=1), wykorzystujący alfa–blending. Oto fragment Vertex Shadera dla tego przebiegu:

```
Out.Fog = FogColor;
Out.Fog.a = FogFactors.x * Out.Pos.z + FogFactors.y;
```

Pixel Shader przepisuje tylko na wyjście zinterpolowany kolor mgły wraz z kanałem alfa.

**Obliczenia animacji tekstury** Koordynaty tekstury mogą być modyfikowane przez dodatkową macierz pamiętaną w encji, co pozwala na animowanie (np. przewijanie) tekstury na powierzchni modelu. Aby uzyskać ten efekt, shader główny otrzymuje makro TEXTURE_ANIMATION=1 oraz macierz tekstury przekazaną w parametrze TextureMatrix. Kod Vertex Shadera przekształcający koordynaty tekstury wygląda następująco:

```
#if (TEXTURE_ANIMATION == 1)
  Out.Tex = mul(float4(In.Tex, 0, 1), (float4x2)TextureMatrix);
#else
  Out.Tex = In.Tex;
#endif
```

**Obliczenia przebiegu bazowego** Wszystkie fragmenty geometrii, które używają renderowania wieloprzebiegowego, są najpierw renderowane przebiegiem bazowym (PASS=0). Wykonuje on kilka czynności. Jedną z nich jest zapis głębokości do Z–bufora. Następne przebiegi wykonują już tylko test Z, bez zapisywania do Z–bufora. Ciekawą optymalizacją jest też wykonywanie testu Alfa tylko w przebiegu bazowym, by w następnych przezroczyste piksele były odrzucane przez test Z dzięki ustawieniu dla takich przypadków funkcji porównującej testu Z na D3DCMP_EQUAL.

Niektóre źródła postulują wypełnianie Z–bufora w osobnym przebiegu z wyłączonym zapisem pikseli do celu renderowania, co ma przyspieszać renderowanie dwukrotnie. W omawianym silniku są jednak efekty, które muszą być wyrenderowane jako „bazowe", więc rozdzielenie ich rysowania od wypełnienia Z–bufora kosztowałoby dodatkowy przebieg, a więc konieczność dodatkowego rysowania całej widocznej geometrii sceny, co zdaniem autora mogłoby zniweczyć potencjalne korzyści wydajnościowe tej sztuczki.

Pierwszym z efektów renderowanych do celu renderowania przez przebieg bazowy jest normalny kolor/tekstura potraktowana jako oświetlenie otoczenia (ang. *Ambient*), czyli bez wpływu konkretnego światła. Makro AMBIENT_MODE może przyjmować wartości: 0 oznaczająca brak oświetlenia otoczenia (kolor jest czarny), 1 oznaczająca pełne oświetlenie otoczenia (stosowane kiedy dany fragment nie podlega oświetleniu lub oświetlenie w silniku jest całkowicie wyłączone), 2 oznaczająca mnożenie koloru podstawowego przez kolor i intensywność oświetlenia otoczenia. Odpowiedni kod Pixel Shadera to:

```
float4 MyDiffuseColor;
#if (AMBIENT_MODE == 0)
  MyDiffuseColor = half4(0, 0, 0, 1);
  #if (ALPHA_TESTING == 1)
    MyDiffuseColor.a = tex2D(DiffuseSampler, In.Tex).a;
  #endif
  Out = MyDiffuseColor;
#else
  #if (DIFFUSE_TEXTURE == 1)
    MyDiffuseColor = tex2D(DiffuseSampler, In.Tex);
  #else
    MyDiffuseColor = DiffuseColor;
  #endif
  Out = ProcessTeamColor(MyDiffuseColor);
#endif
#if (AMBIENT_MODE == 2)
  Out *= AmbientColor;
#endif
```

Drugim efektem przebiegu bazowego jest nakładanie dodatkowej tekstury „emisji" (ang. *Emissive*). Jest to tekstura nie podlegająca oświetleniu, na której zaznaczone są miejsca mające wyglądać jakby same świeciły, np. diody na panelu komputera czy innej maszyny. Włączeniem efektu *Emissive* steruje makro `EMISSIVE`, a tekstura emisji jest przekazana przez parametr `EmissiveTexture`.

```
#if (EMISSIVE == 1)
  Out += tex2D(EmissiveSampler, In.Tex);
#endif
```

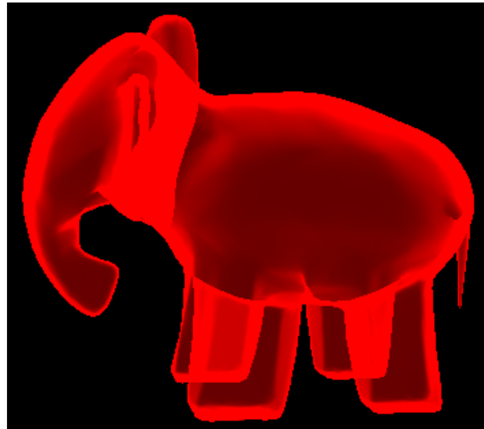
Kolejnym efektem jest mapowanie środowiskowe (ang. *Enivironmental Mapping*). Polga ono na teksturowaniu obiektu mapą sześcienną za pomocą wektora kamery odbitego od powierzchni obiektu. Jeśliby na przykład wyrenderować do takiej mapy sześciennej obraz sceny z pozycji obiektu i nałożyć ją na ten obiekt, to zastosowanie mapowania środowiskowego sprawi, że będzie on wyglądał jak wykonany z metalu odbijającego światło. Ponadto nałożenie specjalnie przygotowanych tekstur pozwala uzyskać wiele ciekawych efektów. Włączeniem tego efektu steruje makro `ENVIRONMENTAL_MAPPING`, a używana do niego tekstura sześcienna przekazywana jest przez parametr `EnvironmentalTexture`. Odpowiedni kod to:

```
////// VERTEX SHADER
#if (ENVIRONMENTAL_MAPPING == 1)
  float3 VecFromCam = (float3)InputPos - CameraPos;
  float3 Reflected = reflect(VecFromCam, InputNormal);
  Out.ReflectedEnvDir = Reflected;
#endif

////// PIXEL SHADER
#if (ENVIRONMENTAL_MAPPING == 1)
  float4 EnvSample = texCUBE(
    EnvironmentalSampler, In.ReflectedEnvDir);
  Out += EnvSample * EnvSample.a;
#endif
```

Ostatnim z efektów przebiegu bazowego jest *Fresnel Term*. Wyliczany jest on również, podobnie jak pozostałe efekty opisane tutaj, niezależnie od oświetlenia konkretnym światłem. Polega na „podświetleniu" obiektu jakby od tyłu dodając określony

kolor tym intensywniej, im bardziej dane miejsce powierzchni jest równoległe do kierunku kamery. Implementacja działająca także dla obiektów dwustronnych i półprzezroczystych pozwala uzyskać ciekawe efekty, jak widać na rys. 3.5. Włączeniem tej techniki steruje makro `FRESNEL_TERM`. Przekazywane w związku z nią parametry to `FresnelColor` (kolor) i `FresnelPower` (wykładnik potęgi). Oto odpowiedni kod głównego shadera:



Rys. 3.5. Przykład zastosowania efektu *Fresnel Term* wraz z materiałem dwustronnym półprzezroczystym do otrzymania wyglądu „ducha".

```
half4 CalcFresnel(float3 VertexPos, float3 VertexNormal)
{
  float3 DirToCam = normalize(CameraPos - VertexPos);
  float Dot = dot(DirToCam, VertexNormal);
  return FresnelColor * pow(1 - abs(Dot), FresnelPower);
}

////// VERTEX SHADER
#if (FRESNEL_TERM == 1)
  Out.Fresnel = CalcFresnel(InputPos, InputNormal);
#endif

////// PIXEL SHADER
#if (FRESNEL_TERM == 1)
  Out += In.Fresnel;
#endif
```

## 3.3. Format pliku modeli 3D

Kształ modeli 3D, mapowanie tekstury i animacje tworzyć trzeba w odpowiednim programie graficznym, np. Blender, 3ds Max, Maya. Powstaje problem z formatem pliku, w którym taki model wczytywany powinien być przez silnik. Istnieje wiele popularnych formatów (np. OBJ, 3DS, MD2, MD5), ale każdy z nich zależnie od potrzeb okazać się może niewystarczający lub po prostu nieodpowiedni. Istnieje też format mający ustandardyzować wymianę danych graficznych 3D między programami — COLLADA. Niestety jest to format bardzo skomplikowany i oparty na XML–u. Dlatego częstą praktyką jest projektowanie własnych formatów plików modeli i pisanie wtyczek dla programów graficznych 3D, eksportujących modele do takiego formatu.

Tak też postąpił autor projektując format nazwany QMSH (od *The Final Quest Mesh*). Odpowiednia wtyczka do programu Blender eksportuje model do tekstowego formatu pośredniego QMSH.TMP, a konsolowe narzędzie Tools przetwarza go na format docelowy QMSH. Jest to format binarny zaprojektowany tak, aby tablicę wierzchołków i indeksów można było wczytać wprost do buforów Direct3D, bez żadnego przetwarzania.

Nagłówek pliku składa się z następujących pól:

- identyfikator formatu — 6 znaków `"TFQMSH"`,
- identyfikator wersji — 2 znaki `"10"`,
- dlagi — 1 bajt, w którym: ustawiony bit `0x01` oznacza, że wierzchołki posiadają wektory styczne *Tangent* i *Binormal*, ustawiony bit `0x02` oznacza, że obecny jest szkielet, animacje, a wierzchołki posiadają informację o wpływie kości,
- liczba wierzchołków — typu `uint2`,
- liczba trójkątów — typu `uint2`,
- liczba pod–siatek — typu `uint2`,
- liczba kości — typu `uint2`,
- liczba animacji — typu `uint2`,
- promień sfery otaczającej siatkę, której środek leży w punkcie $(0, 0, 0)$ — typu `float`,
- minimalne i maksymalne współrzędne prostopadłościanu otaczającego siatkę $(x_1, y_1, z_1), (x_2, y_2, z_2)$ — 6 liczb typu `float`,
- offset od początku pliku do początku bufora wierzchołków — typu `uint4`,
- offset od początku pliku do początku bufora indeksów — typu `uint4`,
- offset od początku pliku do początku danych o pod–siatkach — typu `uint4`,
- offset od początku pliku do początku danych kości — typu `uint4` (0 jeśli brak),
- offset od początku pliku do początku danych animacji — typu `uint4` (0 jeśli brak).

Bufor wierzchołków rozpoczyna się bajtem kontrolnym o wartości 0. Każdy wierzchołek składa się z pól:

- pozycja $(x, y, z)$ — 3 liczby typu `float`,
- informacje o wypływie kości, obecne tylko jeśli ustawiona jest odpowiednia flaga w nagłówku:
  - waga wpływu na dany wierzchołek pierwszej z dwóch kości — liczba typu `float` w zakresie $[0; 1]$. Waga wpływu drugiej kości jest wyliczana automatycznie i wynosi $w_2 = 1 - w_1$,
  - liczba 32–bitowa bez znaku, której najmłodszy bajt przechowuje indeks pierwszej kości wpływającej na ten wierzchołek, a drugi z kolei bajt — indeks drugiej kości,
- wektor normalny $(x, y, z)$ — 3 liczby typu `float`. Jest zawsze znormalizowany,

- koordynaty tekstury $(t_x, t_y)$ — 2 liczby typu `float`,
- wektory styczne, obecne tylko jeśli ustawiona jest odpowiednia flaga w nagłówku. $(x_t, y_t, z_t), (x_b, y_b, z_b)$ — 6 liczb typu `float`.

Bufor indeksów rozpoczyna się bajtem kontrolnym o wartości 0. Każdemu trójkątowi odpowiadają w nim 3 indeksy wierzchołków. Każdy indeks to liczba typu `uint2`.

Tablica informacji o pod–siatkach rozpoczyna się bajtem kontrolnym o wartości 0. Każda pod–siatka jest opisana następującą strukturą:

- numer pierwszego trójkąta pod–siatki — typu `uint2`,
- liczba trójkątów pod–siatki — typu `uint2`,
- najniższy indeks używanego przez pod–siatkę wierzchołka — typu `uint2`,
- liczba wierzchołków, która razem z poprzednim polem wyznacza zakres wierzchołków używanych przez pod–siatkę — typu `uint2`,
- nazwa pod–siatki — 1 bajt długości + znaki ASCII,
- nazwa materiału — 1 bajt długości + znaki ASCII.

Tablica kości rozpoczyna się bajtem kontrolnym o wartości 0. Każda kość jest opisana następującą strukturą:

- indeks kości nadrzędnej — typu `uint2`. Kości indeksowane są od 1, warość 0 oznacza brak kości,
- macierz kości $4 \times 3$ przekształcająca współrzędne z układu lokalnego tej kości do układu kości nadrzędnej (lub układu modelu, jeśli to kość pierwszego poziomu), zapisana kolejnymi wierszami — 12 liczb typu `float`,
- nazwa kości — 1 bajt długości + znaki ASCII.

Tablica animacji rozpoczyna się bajtem kontrolnym o wartości 0. Każda animacja jest opisana następującą strukturą:

- nazwa animacji — 1 bajt długości + znaki ASCII,
- długość animacji — liczba typu `float`, w sekundach,
- liczba klatek kluczowych — typu `uint2`,
- tablica klatek kluczowych, a każda posiada pola:
  - czas — typu `float`,
  - tablica przekształceń. Przekształcenie dla każdej kości posiada pola:
    - translacja — wektor $(x, y, z)$, 3 liczby typu `float`,
    - rotacja — kwaternion $(x, y, z, w)$, 4 liczby typu `float`,
    - skalowanie — jedna liczba typu `float`.

## 3.4. Implementacja animacji szkieletowej

Animacja szkieletowa polega na deformacji siatki modelu. Wierzchołki (ich pozycja, jak i wektory normalne oraz styczne) są przekształcane przez „kości". Kość nie jest żadnym widocznym obiektem, a jedynie abstrakcyjnym pojęciem reprezentującym określone dane matematyczne. Dzięki nim całe grupy wierzchołków (np. wszelka geometria przedstawiająca rękę) może się poruszać sterowana pojedynczym przekształceniem opisującym kość tej ręki, przesuwając, skalując oraz przede wszystkim obracając się wokół punktu początkowego danej kości. Kości tworzą hierarchię, tak że ich przekształcenia są składane. Przykładowo, część siatki sterowana przez kość dłoni porusza się też razem z całym ramieniem.

*Skinning* to technika, w której na wierzchołek może wypływać więcej niż jedna kość. Wektory takiego wierzchołka są wówczas przekształcane przez macierze wszystkich kości, które na niego wpływają, a następnie interpolowane stosownie do wag wpływu tych kości. Wierzchołek musi więc mieć zapisane indeksy kości, które na niego wpływają oraz wagi wpływu tych kości. Autor zdecydował się na umożliwienie wypływu 2 kości na wierzchołek.

Obliczenia animacji szkieletowej składają się z trzech etapów. Pierwszy etap jest wykonywany tylko raz, podczas pierwszego zapytania. Wylicza on tablicę macierzy, które dla każdej kości opisują przekształcenie ze współrzędnych modelu do współrzędnych tej kości. Obliczenia te realizuje metoda `GetModelToBoneMatrices` klasy `res::QMesh`, a algorytm w pseudokodzie wygląda następująco:

```
ModelToBoneMatrices = MATRIX[]
ModelToBoneMatrices[0] = IDENTITY
foreach (Kość in Kości modelu):
  ModelToBoneMatrices[Kość] = Odwrotność macierzy (Kość.Macierz)
  if (Kość nie jest pierwszego poziomu):
    ModelToBoneMatrices[Kość] =
      ModelToBoneMatrices[Kość.Kość_nadrzędna] *
      ModelToBoneMatrices[Kość]
return ModelToBoneMatrices
```

Drugi etap polega na wyliczeniu tablicy tzw. macierzy kości. Wykonuje go metoda `GetBoneMatrices` klasy `res::QMesh`. Dla każdej kości powstaje macierz, która reprezentuje przekształcenie wierzchołków z ich pozycji spoczynkowej w przestrzeni lokalnej modelu do pozycji ustalonej przez daną animację, również w przestrzeni modelu. Ponieważ zbudowanie tych macierzy jest dość kosztowne obliczeniowo, ostatnio wyliczone tablice macierzy są zapamiętywane w liście struktur typu `BoneMatrixCacheEntry`. Procedura wyliczania macierzy kości dla podanej animacji i jej czasu (lub pary animacji i współczynnika interpolacji między nimi) wygląda w pseudokodzie następująco:

```
BoneToParentPoseMat = MATRIX[]
BoneToParentPoseMat[0] = IDENTITY
if (Jedna animacja):
  foreach (Kość in Kości modelu):
```

```
    float Scal = Animacja.Skalowanie(Kość, Czas)
    MATRIX ScalMat = Macierz skalowania (Scal)
    QUATERNION Rot = Animacja.Rotacja(Kość, Czas)
    MATRIX RotMat = Macierz rotacji (Rot)
    VEC3 Trans = Animacja.Translacja(Kość, Czas)
    MATRIX TransMat = Macierz translacji (Trans)
    BoneToParentPoseMat[Kość] =
      ScalMat * RotMat * TransMat * Kość.Macierz
else: // Interpolacja między dwiema animacjami
  foreach (Kość in Kości modelu):
    float Scal1 = Animacja1.Skalowanie(Kość, Czas1)
    float Scal2 = Animacja2.Skalowanie(Kość, Czas2)
    float Scal = LERP od Scal1 do Scal2 wg LerpT
    MATRIX ScalMat = Macierz skalowania (Scal)
    QUATERNION Rot1 = Animacja1.Rotacja(Kość, Czas1)
    QUATERNION Rot2 = Animacja2.Rotacja(Kość, Czas2)
    QUATERNION Rot = SLERP od Rot1 do Rot2 wg LerpT
    MATRIX RotMat = Macierz rotacji (Rot)
    VEC3 Trans1 = Animacja1.Translacja(Kość, Czas1)
    VEC3 Trans2 = Animacja2.Translacja(Kość, Czas2)
    VEC3 Trans = LERP od Trans1 do Trans2 wg LerpT
    MATRIX TransMat = Macierz translacji (Trans)
    BoneToParentPoseMat[Kość] =
      ScalMat * RotMat * TransMat * Kość.Macierz

BoneToModelPoseMat = MATRIX[]
BoneToModelPoseMat[0] = IDENTITY
foreach (Kość in Kości modelu):
  if (Kość jest pierwszego poziomu):
    BoneToModelPoseMat[Kość] = BoneToParentPoseMat[Kość]
  else:
    BoneToModelPoseMat[Kość} = BoneToParentPoseMat[Kość] *
      BoneToModelPoseMat[Kość.Kość_nadrzędna]

BoneMatrices = MATRIX[]
BoneMatrices[0] = IDENTITY
foreach (Kość in Kości modelu):
  BoneMatrices[Kość] = ModelToBoneMatrices[Kość] *
    BoneToModelPoseMat[Kość]
return BoneMatrices
```

Każda z tablic zawiera w pierwszym elemencie macierz identycznościową, ponieważ indeks 0 oznacza brak kości. Właściwe kości indeksowane są od 1. Dzięki temu wybrane wierzchołki siatki mogą pozostawać bez wpływu ze strony jakiejkolwiek kości (albo tylko pod częściowym wpływem). Wyliczane najpierw macierze `BoneToParentPoseMat` reprezentują przekształcenia ze współrzędnych danej kości do współrzędnych jej kości nadrzędnej w pozycji zgodnej z żądaną animacją. Z nich powstają następnie macierze `BoneToModelPoseMat`, które reprezentują przekształcenia ze współrzędnych danej kości do współrzędnych modelu, również w ustalonej pozycji. Wreszcie, ostateczne macierze `BoneMatrices` reprezentują, jak już zostało napisane wyżej, przekształcenie z pozycji spoczynkowej do pozycji ustalonej przez daną animację w przestrzeni modelu.

Trzeci etap jest wykonywany w czasie rzeczywistym i polega na zastosowaniu macierzy kości do każdego wierzchołka siatki. Wierzchołki są zapisane w przestrzeni modelu, w pozycji spoczynkowej (ang. *Bind Pose*). Macierze kości przekształcają z tego układu do układu w przestrzeni modelu, ale w pozycji ustalonej przez daną animację.

Tak więc animowanie modelu polega na pomnożeniu danych wierzchołka przez macierze kości, które na niego wpływają oraz zinterpolowaniu wyniku stosownie do wag wpływu tych kości. Kod Vertex Shadera, który to wykonuje, wygląda następująco:

```
#if (SKINNING == 1)
float3 SkinPos(float3 Pos, VS_INPUT Input)
{
  float3 OutputPos =
    mul(float4(Pos, 1), BoneMatrices[Input.BoneIndices[0]]) *
    Input.BoneWeight0;
  OutputPos +=
    mul(float4(Pos, 1), BoneMatrices[Input.BoneIndices[1]]) *
    (1 - Input.BoneWeight0);
  return OutputPos;
}
half3 SkinNormal(half3 Normal, VS_INPUT Input)
{
  half3 OutputNormal =
    mul(Normal, (half3x3)BoneMatrices[Input.BoneIndices[0]]) *
    Input.BoneWeight0;
  OutputNormal +=
    mul(Normal, (half3x3)BoneMatrices[Input.BoneIndices[1]]) *
    (1 - Input.BoneWeight0);
  return OutputNormal;
}
#endif

////// VERTEX SHADER
float3 InputPos;
half3 InputNormal;
half3 InputTangent;
half3 InputBinormal;
#if (SKINNING == 1)
  InputPos = SkinPos(In.Pos, In);
  InputNormal = SkinNormal(In.Normal, In);
  InputTangent = SkinNormal(In.Tangent, In);
  InputBinormal = SkinNormal(In.Binormal, In);
#else
  InputPos = In.Pos;
  InputNormal = In.Normal;
  InputTangent = In.Tangent;
  InputBinormal = In.Binormal;
#endif
```

Klasa `QMesh` potrafi wykonywać analogiczny algorytm na CPU dla potrzeb liczenia kolizji promienia z animowanym modelem. Odpowiedni kod znajduje się w metodzie `RayCollision_Bones`.

## 3.5. Implementacja efektów cząsteczkowych

Efekt cząsteczkowy, dostępny w silniku jako encja typu `ParticleEntity`, jest typu stanowego [57]. Bufor wierzchołków jest wypełniany tylko raz, podczas inicjalizacji, a w czasie rzeczywistym parametry cząstek są wyliczane na GPU przez shader, którego kod mozna znaleźć w pliku `ParticleShader.fx`. Podczas tworzenia obiektu trzeba podać nazwę zasobu tekstury, tryb alfa–blendingu, liczbę cząstek oraz wypełnioną

strukturę `PARTICLE_DEF`. Struktura ta opisuje zmianę parametrów cząstek w czasie i składa się z następujących pól:

```
struct PARTICLE_DEF {
  uint CircleDegree;
  VEC3 PosA2_C, PosA2_P, PosA2_R;
  VEC3 PosA1_C, PosA1_P, PosA1_R;
  VEC3 PosA0_C, PosA0_P, PosA0_R;
  VEC3 PosB2_C, PosB2_P, PosB2_R;
  VEC3 PosB1_C, PosB1_P, PosB1_R;
  VEC3 PosB0_C, PosB0_P, PosB0_R;
  VEC4 ColorA2_C, ColorA2_P, ColorA2_R;
  VEC4 ColorA1_C, ColorA1_P, ColorA1_R;
  VEC4 ColorA0_C, ColorA0_P, ColorA0_R;
  float OrientationA1_C, OrientationA1_P, OrientationA1_R;
  float OrientationA0_C, OrientationA0_P, OrientationA0_R;
  float SizeA1_C, SizeA1_P, SizeA1_R;
  float SizeA0_C, SizeA0_P, SizeA0_R;
  float TimePeriod_C, TimePeriod_P, TimePeriod_R;
  float TimePhase_C, TimePhase_P, TimePhase_R;
};
```

Pola oznaczone `A0`, `A1`, `A2`, `B0`, `B1`, `B2` to parametry równania:

$$v = A_2 \cdot t^2 + A_1 \cdot t + A_0 + B_2 \cdot \sin(B_1 \cdot t + B_0) \tag{3.1}$$

`Pos` to pozycja cząstki, w przestrzeni lokalnej obiektu. `Color` to kolor cząstki wraz z przezroczystością w kanale Alfa. `Orientation` to orientacja cząstki, czyli jej kąt obrotu w radianach wokół kierunku patrzenia kamery. `Size` to rozmiar cząstki, w przestrzeni lokalnej modelu. `Period` to okres, czyli czas w sekundach, po którym cząstka powtarza swój bieg od początku. `Phase` to faza, czyli przesunięcie czasu $[0; 1]$, dzięki któremu różne cząstki mogą być w danej chwili w różnym miejscu od swojej pozycji początkowej do końcowej.

Pola z przyrostkiem `C` (od *Const*) oznaczają składnik stały parametru. Pola z przyrostkiem `P` (od *Particle Number*) oznaczają składnik parametru mnożony przez liczbę $[0; 1]$ zależną od numeru cząstki. Pola z przyrostkiem `R` (od *Random*) oznaczają składnik parametru mnożony przez zapamiętaną dla cząstki liczbę losową $[0; 1]$.

Trzeba przyznać, że sterowanie tak zdefiniowanymi parametrami efektu cząsteczkowego jest trudne, szczególnie dla nieprogramisty. Trudno też wyobrazić sobie sposób, w jaki mogłaby być umożliwiona ich wizualna edycja w specjalnym edytorze. Jednak przy odrobinie wprawy i z pomocą narzędzia wyliczającego współczynniki równania liniowego i kwadratowego na podstawie podanych punktów, otrzymywanie pożądanych rezultatów jest możliwe. W zamian, tak zaprojektowany system oferuje bardzo dużą elastyczność i szerokie możliwości sterowania zachowaniem cząstek przy jednoczesnej wysokiej wydajności. W sumie wzory obliczające parametry danej cząstki można zapisać w pseudokodzie następująco:

```
TimePeriod = TimePeriod_C + TimePeriod_P * P + TimePeriod_R * R
TimePhase = TimePhase_C + TimePhase_P * P + TimePhase_R * R
t = frac(GlobalTime / TimePeriod + TimePhase)
```

```
PosA2 = PosA2_C + PosA2_P * P + PosA2_R * R
PosA1 = PosA1_C + PosA1_P * P + PosA1_R * R
PosB2 = PosB2_C + PosB2_P * P + PosB2_R * R
PosB1 = PosB1_C + PosB1_P * P + PosB1_R * R
PosB0 = PosB0_C + PosB0_P * P + PosB0_R * R
ColorA2 = ColorA2_C + ColorA2_P * P + ColorA2_R * R
ColorA1 = ColorA1_C + ColorA1_P * P + ColorA1_R * R
ColorA0 = ColorA0_C + ColorA0_P * P + ColorA0_R * R
OrientationA1 = OrientationA1_C + OrientationA1_P * P +
  OrientationA1_R * R
OrientationA0 = OrientationA0_C + OrientationA0_P * P +
  OrientationA0_R * R
SizeA1 = SizeA1_C + SizeA1_P * P + SizeA1_R * R
SizeA0 = SizeA0_C + SizeA0_P * P + SizeA0_R * R

Pos = PosA2 * t^2 + PosA1 * t + PosA0 +
  PosB2 * sin(PosB1 * t + PosB0)
Color = ColorA2 * t^2 + ColorA1 * t + ColorA0
Orientation = OrientationA1 * t + OrientationA0
Size = SizeA1 * t + SizeA0
```

Struktura wierzchołka to:

```
struct PARTICLE_VERTEX
{
  VEC3 StartPos;         // Semantyka D3DFVF_XYZ
  COLOR VertexFactors;   // Semantyka D3DFVF_DIFFUSE
  VEC2 ParticleFactors;  // Semantyka D3DFVF_TEXCOORDSIZE2(0)
}
```

Kreatywne wykorzystanie pól wierzchołka do celów innych niż oryginalnie przewidziane jest typowe dla zaawansowanych efektów realizowanych z użyciem shaderów. W powyższej strukturze pole `StartPos` wyznacza pozycję początkową cząstki. Pole `VertexFactors` przechowuje w swoich składowych RGB kolejno: R — kąt charakterystyczny dla wierzchołka cząstki, mapowany przez Vertex Shader z przedziału $[0;1]$ do $[1\pi/4;7\pi/4]$, G — współrzędna tekstury $t_x$, równa 0 lub 1, B — współrzędna tekstury $t_y$, równa 0 lub 1. Wykorzystanie semantyki koloru *Diffuse* pozwoliło zmieścić te trzy wartości do pojedynczej liczby 32–bitowej, co oszczędza pamięci zapewniając precyzję wystarczającą do tych zastosowań. Warto dodać, że pola tego typu są w kodzie C++ bajtami o wartościach $[0;255]$, ale w kodzie shadera HLSL stają się liczbami zmiennoprzecinkowymi $[0.0;1.0]$. Pole `ParticleFactors` w komponencie `x` przechowuje współczynnik charakterystyczny cząstki $[0;1]$ zależny liniowo od numeru cząstki (`P`), a w komponencie `y` przechowuje współczynnik losowy cząstki $[0;1]$ (`R`).

Współczynnik losowy cząstki `R` nie może być pojedynczy, ponieważ wówczas, jeśli na przykład od niego zależałaby zarówno pozycja $y$ jak i kolor, cząstki znajdujące się wyżej były równocześnie jaśniejsze, co zepsułoby wizualnie efekt losowości. Dlatego Vertex Shader efektu cząsteczkowego generuje na podstawie współczynnika losowego, pobranego ze struktury wierzchołka, trzy współczynniki, które wykorzystuje potem w różnej kolejności. Odpowiedni kod HLSL to:

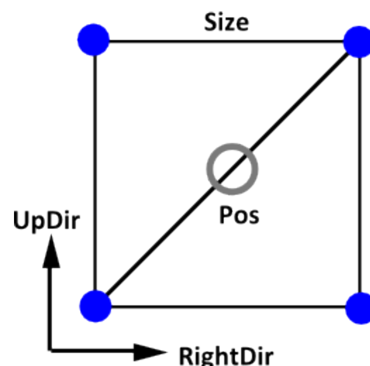
```
float ParticleP = In.ParticleFactor.x;
float3 ParticleR = frac(In.ParticleFactor.yyy *
```

```
  float3(324.2135, 231.3542, 147.4534));

////// Jedna z instrukcji wykorzystujących te współczynniki:
float4 ColorA1 = Data[3] + Data[4] * ParticleP +
  Data[5] * ParticleR.yzxy;
```

Cząsteczki są rysowane jako kwadraty zbudowane z dwóch trójkątów, zwrócone zawsze przodem do kamery. Istnieje wprawdzie mechanizm sprzętowego zamieniania pojedynczych wierzchołków na takie kwadraty — *Point Sprites* — jest nie został on tutaj użyty, gdyż do tak zaawansowanego systemu cząsteczkowego jest niewystarczający. Nie ma on możliwości obracania cząstek, a na kartach bez `D3DFVFCAPS_PSIZE` (m.in. GeForce FX) także zmiany wielkości poszczególnych cząstek.

Dlatego opisany tutaj efekt cząsteczkowy jawnie podaje po 4 wierzchołki i 6 indeksów na każdą cząstkę, by z nich zbudować taki kwadrat. Parametry cząstki muszą być od nowa wyliczone dla każdego z jej wierzchołków osobno (tak działa Vertex Shader). Dlatego wszystkie wierzchołki danej cząstki mają zapisane te same parametry, z wyjątkiem opisanego wyżej pola `VertexFactors`, które pozwala je rozróżnić „rozsuwając” wierzchołki od pozycji środka cząstki w kierunku zgodnym z wektorami „w prawo” i „w górę” pobranymi z kamery oraz nakładając odpowiednie teksturowanie, tak jak to pokazuje rysunek 3.6. Tych czynności dokonują kończące instrukcje Vertex Shadera:



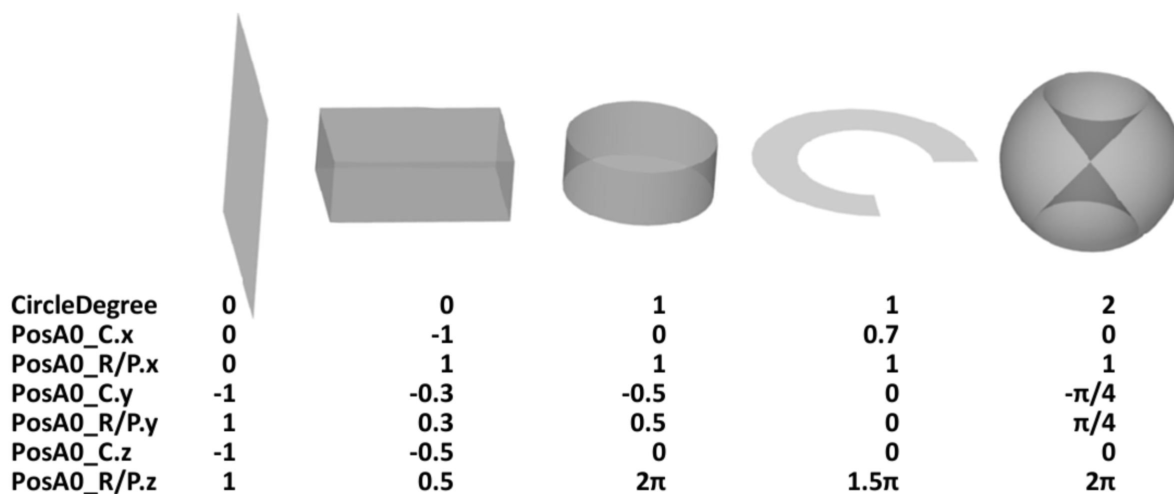

Rys. 3.6. Kwadrat przedstawiający cząsteczkę, zbudowany z 4 wierzchołków odsuwanych od pozycji środka cząsteczki zgodnie z wektorami kamery „w prawo” i „w górę”.

```
float HalfSizeDiagonal = Size * (1.4142135623730950488O / 2);
// Mapowanie 0 .. 1 na 1/4*PI .. 7/4*PI
// Wzór który to robi: y = 3/2*PI * x + 1/4*PI
float Angle = In.VertexFactors.r * 4.71238898 + 0.785398163 +
  Orientation;

Pos = Pos + (RightDir * cos(Angle) + UpDir * sin(Angle)) *
  HalfSizeDiagonal;
Out.Pos = mul(float4(Pos, 1), WorldViewProj);
Out.Color = Color;
Out.Tex = float2(In.VertexFactors.g, In.VertexFactors.b);
```

Pozycja początkowa cząstek jest wyliczana na CPU tylko raz, podczas tworzenia encji efektu cząsteczkowego i zostaje zapisana do wierzchołków. Dlatego może być wyliczana wg bardziej skomplikowanych zależności. Zostało to wykorzystane do umożliwienia generowania pozycji początkowej cząstek z obszaru o różnym kształcie. Steruje

tym pole `CircleDegree` struktury `PARTICLE_DEF`. Jeśli wynosi ono 0, współczynniki `A0` posiadają dosłowną interpretację, wyznaczając zakres $[(x_1, y_1, z_1); (x_2, y_2, z_2)]$, z którego mają być generowane pozycje cząstek. Pozwala to otrzymać obszar generowania w kształcie punktu, odcinka, prostokąta lub prostopadłościanu. Jeśli `CircleDegree` jest równe 1, pozycje początkowe są generowane z okręgu, koła lub walca o podstawie równoległej do osi $XZ$. Składowe $y$ parametru `A0` wyznaczają zakres $y_1 \ldots y_2$, składowe $x$ opisują promień, a składowe $z$ kąt. Jeśli natomiast `CircleDegree` jest równe 2, emiter cząstek ma kształt kuli, składowe $x$ opisują promień, składowe $y$ pierwszy kąt (długość geograficzną), a składowe $z$ drugi kąt (szerokość geograficzną). Przykłady pokazuje rys. 3.7.



| | | | | | |
|---|---|---|---|---|---|
| CircleDegree | 0 | 0 | 1 | 1 | 2 |
| PosA0_C.x | 0 | -1 | 0 | 0.7 | 0 |
| PosA0_R/P.x | 0 | 1 | 1 | 1 | 1 |
| PosA0_C.y | -1 | -0.3 | -0.5 | 0 | -π/4 |
| PosA0_R/P.y | 1 | 0.3 | 0.5 | 0 | π/4 |
| PosA0_C.z | -1 | -0.5 | 0 | 0 | 0 |
| PosA0_R/P.z | 1 | 0.5 | 2π | 1.5π | 2π |

Rys. 3.7. Przykładowe kształty obszarów, z których wybrane mogą być pozycje początkowe cząsteczek, zależnie od parametrów struktury `PARTICLE_DEF`.

Przykładowe efekty działania systemu cząsteczkowego prezentuje zrzut ekranu 4.8. W efekcie pierwszym od lewej kolory cząstek zależą od współczynnika losowego `R`, ale ich pozycja jest opisana wyłącznie z użyciem współczynnika `P`, zależnego od numeru cząstki. Dzięki temu poszczególne cząstki są od siebie umieszczone w równych odstępach. Wszystkie mają tą samą pozycję początkową i poruszają się po tej samej krzywej w kształcie spirali. Każda jest w innym miejscu krzywej dzięki uzależnieniu fazy od numeru cząstki: `TimePhase_C=0`, `TimePhase_P=1`. Z kolei efekt ognia widoczny po prawej stronie tworzą cząstki, których parametry (szczególnie pozycja i tor ruchu) zależą od współczynnika losowego `R`, co daje wrażenie przypadkowości.

## 3.6. Implementacja efektów postprocessingu

Proste efekty postprocessingu, takie jak zamalowanie całego ekranu na podany kolor (klasa `PpColor`) czy podaną teksturą (klasa `PpTexture`) nie wymagają dokładnego omówienia. Ich realizacja polega na narysowaniu prostokąta pokrywającego cały obszar ekranu (ang. *Fullscreen Quad*) z włączonym alfa–blendingiem. Warto dodać, że

takie proste technicznie efekty mogą być bardzo ważne. Na przykład na początku i przy zakończeniu gry ekran może się płynnie rozjaśniać/ściemniać do czarnego, a podczas otrzymywania obrażeń przez bohatera może się robić czerwony. Innym przykładem takiego prostego efektu jest losowe przesuwanie („trzęsienie”) kamery podczas eksplozji.

**Realizacja efektu funkcji** Efekt zaimplementowany w klasie `PpFunction` pozwala na przekształcenie wszystkich pikseli rysowanego obrazu za pomocą pewnej funkcji, której współczynniki można zmieniać. Funkcję tą można przedstawić w pseudokodzie jako:

```
color Kolor_grayscale.rgb = Odcień szarości (Kolor_we)
Kolor_wy = LERP od Kolor_we do Kolor_grayscale wg GrayscaleFactor
Kolor_wy = Kolor_wy * A_factor + B_factor
return Kolor_wy
```

Trzy współczynniki sterujące tą funkcję zapewniają bardzo szeroką gamę możliwości, na przykład:

- `GrayscaleFactor`$= 0$, `A_factor`$= (v, v, v)$, `B_factor`$=(0, 0, 0)$ pozwala regulować jasność,
- `GrayscaleFactor`$= 0$, `A_factor`$= (v, v, v)$, `B_factor`$= (0.5 - 0.5v, 0.5 - 0.5v, 0.5 - 0.5v)$ pozwala regulować kontrast,
- `GrayscaleFactor`$= 1 - v$, `A_factor`$= (1, 1, 1)$, `B_factor`$= (0, 0, 0)$ pozwala regulować nasycenie,
- `GrayscaleFactor`$= 1$, `A_factor`$= (1, 1, 1)$, `B_factor`$= (0, 0, 0)$ zamienia obraz na odcienie szarości,
- `GrayscaleFactor`$= 1$, `A_factor`$(1, 1, 1)$, `B_factor`$= (0.191, -0.054, -0.221)$ daje efekt sepii,
- `GrayscaleFactor`$= 1$, `A_factor`$= (-1, -1, -1)$, `B_factor`$= (1, 1, 1)$ odwraca kolory,
- `GrayscaleFactor`$= 0$, `A_factor`$= (1, 0, 0)$, `B_factor`$= (0, 0, 0)$ pokazuje tylko kanał czerwony.

Implementacja tego efektu opiera się na przepisaniu obrazu z tekstury, do której renderowana jest scena, do bufora ramki, za pomocą multishadera `PpShader` (patrz rys. 3.8). On wykonuje także inne czynności związane z różnymi efektami postprocessingu, dlatego jest kompilowany na żądanie z takim zestawem możliwości, jakie są w danej chwili potrzebne.

**Realizacja efektu Tone Mapping** Efekt udający *Tone Mapping* z HDR polega na przyciemnianiu lub rozjaśnianiu obrazu, zależnie od jego średniej jasności. Algorytm wygląda następująco:

1. Scena zostaje wyrenderowana do tekstury pomocniczej.




Rys. 3.8. Podczas używania niektórych efektów postprocessingu następuje narysowanie sceny do pomocniczej tekstury, z której dopiero zostaje przerysowana do bufora ramki za pomocą shadera `PpShader`.

2. Metoda `CalcBrightness` klasy `PpToneMapping` kopiuje najpierw tą teksturę do dodatkowej tekstury pomocniczej udostępnianej przez `EngineServices` jako `ST_TONE_MAPPING_VRAM`, która ma wielkość równą 1/50 rozdzielczości ekranu.
3. Zawartość tej tekstury zostaje sprowadzona z karty graficznej do pamięci RAM za pomocą funkcji Direct3D `GetRenderTargetData`.
4. Metoda `MeasureBrightness` próbkuje wybrane miejsca tej tekstury i na tej podstawie szacuje średnią jasność obrazu.
5. Podczas przerysowania obrazu do bufora ramki, `PpShader` reguluje jasność obrazu na podstawie danych pobranych z klasy `PpToneMapping`.

Ponieważ sprowadzanie zawartości tekstury z pamięci karty graficznej do pamięci systemowej jest bardzo niewydajne, klasa robi to tylko co określony czas, a nie w każdej klatce. Ponadto zmiana rejestrowanej jasności jest wygładzana i reaguje z pewnym opóźnieniem, co daje naturalny efekt imitujący adaptację źrenicy oka.

**Realizacja efektu Bloom** Efekt *Bloom* jest częścią imitacji HDR i pozwala przedstawić obszary, które wyglądają jakby były „jaśniejsze niż białe”. Przykład pokazuje rys. 3.9.

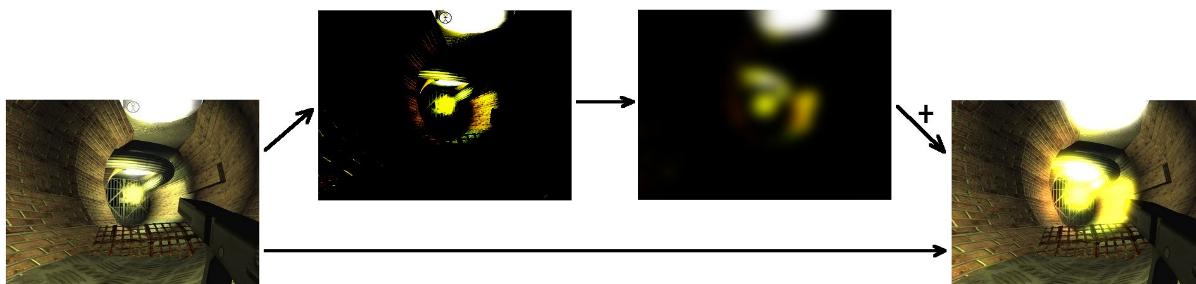


Rys. 3.9. Przykład sporządzony w programie graficznym obrazuje efekt *Bloom* (dwa ostatnie paski).

1. Scena zostaje wyrenderowana do tekstury pomocniczej.
2. Metoda `CreateBloom` z klasy `PpBloom` przerysowuje obraz między dwiema teksturami pomocniczymi, wykonując najpierw przebieg pozostawiający tylko miejsca jasne (*BrightPass*), a następnie dwuetapowe rozmycie filtrem Kawase'a opisanym w [55] (*BlurPass*). Wynik zostaje zachowany w dodatkowej teksturze pomocniczej `ST_BLUR_1`. Tekstury pomocnicze `ST_BLUR_1` i `ST_BLUR_2` są 4 razy mniejsze od rozdzielczości ekranu, co nie tylko oszczędza pamięć i przyspiesza przetwarzanie, ale dzięki filtrowaniu liniowemu powoduje dodatkowe rozmycie.

3. Zawartość tekstury głównej z obrazem zostaje przerysowana do bufora ramki. Ten efekt nie ma wpływu na to przerysowanie.
4. Na obraz w buforze ramki dorysowana zostaje tekstura `ST_BLUR_1` z blendingiem addytywnym.

Cały proces ilustruje rys 3.10.




Rys. 3.10. Proces powstawania efektu *Blooom* jako dodatkowe, addytywne nakładanie na obraz jego przetworzonej wersji.

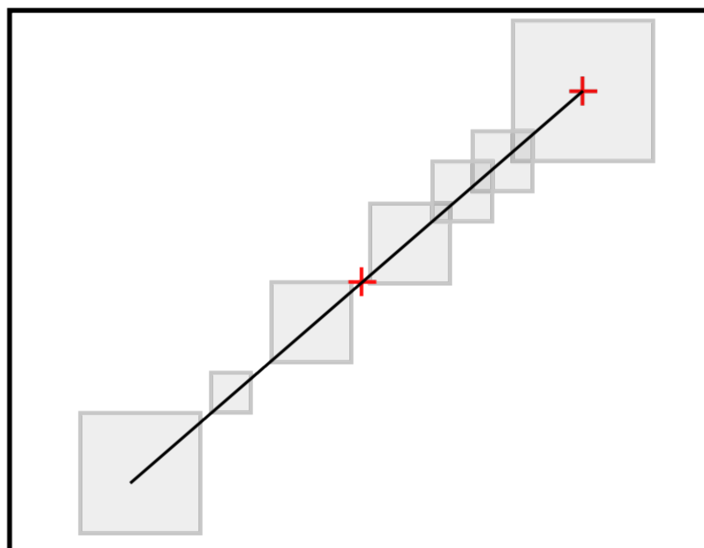
**Realizacja efektu sprzężenia zwrotnego** Efekt sprzężenia zwrotnego został wykonany na podstawie [3]. Polega na narysowaniu na obrazie nowej klatki półprzezroczystej, nieco przekształconej (np. przesuniętej, przeskalowanej, obróconej) wersji klatki poprzedniej. To pozwala uzyskać szereg efektów takich jak prosta imitacja rozmycia ruchu (ang. *Motion Blur*) dla całego ekranu. Algorytm wykonania tego efektu jest następujący:

1. Obraz nowej klatki jest rysowany do bufora ramki. Nieistotne jest przy tym, czy wcześniej był rysowany poprzez pomocniczą teksturę i `PpShader`).
2. Metoda `PpFeedback::Draw` najpierw nakłada na obraz w buforze ramki obraz z klatki poprzedniej, zapamiętanej w dodatkowej teksturze pomocniczej `ST_FEEDBACK`, w postaci półprzezroczystej i odpowiednio przekształconej.
3. Następnie ta sama metoda przerysowuje końcowy obraz ekranu do tekstury `ST_FEEDBACK` celem wykorzystania w następnej klatce.

Tekstura pomocnicza `ST_FEEDBACK` ma rozmiar 2 razy mniejsze od rozdzielczości bufora ramki. To przyspiesza renderowanie i oszczędza pamięci, a zarazem nie daje widocznego zmniejszenia jakości obrazu, ponieważ obraz rysowany z tej tekstury i tak jest półprzezroczysty.

**Realizacja efektu błysku soczewek** Efekt błysku soczewek został wykonany na podstawie [3]. Polega na narysowaniu na obrazie końcowym, w sposób dwuwymiarowy, przygotowanych specjalnie w tym celu różnych tekstur błysków, ułożonych w linii prostej łączącej pozycję Słońca na ekranie z środkiem ekranu, tak jak to ilustruje rys. 3.11.

Algorytm wykonania całości efektu wygląda następująco:



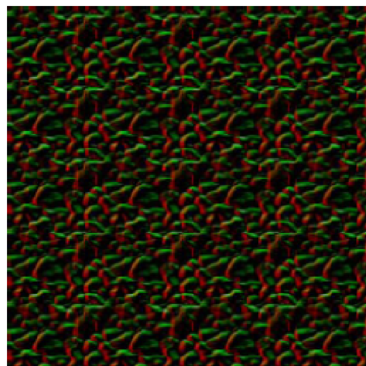
Rys. 3.11. Efekt błysku soczewek jako dwuwymiarowe kwadraty rysowane wzdłuż linii łączącej pozycję Słońca ze środkiem ekranu.

1. Klasa `Scene` po narysowaniu wszystkich obiektów sceny (obojętne czy do tekstury pomocniczej, czy wprost do bufora ramki) sprawdza i zwraca w metodzie `QuerySunVisibleFactor` widoczność Słońca za pomocą zapytania sprzętowego o zasłanianie (ang. *Occlusion Query*).
2. Pod koniec całego procesu rysowania, kiedy już w buforze ramki znajduje się cały obraz 3D, metoda `Draw` klasy `PpLensFlare` dorysowuje półprzezroczyste błyski soczewek.

**Realizacja efektu mgły ciepła** Efekt mgły ciepła (ang. *Heat Haze*) został wykonany na podstawie [56]. Polega on na „falowaniu” obrazu w wyznaczonych miejscach, które mają imitować rozgrzane powietrze, np. nad ogniskiem czy płomieniem świecy. Jego algorytm wygląda następująco:

1. Na początku renderowania sceny, metoda `CreateDrawData` wyznacza zbiór „encji ciepła” typu `HeatEntity` widocznych w zasięgu kamery.
2. Scena jest renderowana do tekstury pomocniczej (nie bezpośrednio do bufora ramki).
3. Kanał Alfa tekstury pomocniczej zostaje wyzerowany. Direct3D nie posiada możliwości czyszczenia tylko wybranych kanałów w metodzie `Clear`, tak więc zostało to zrealizowane poprzez narysowanie pełnoekranowego prostokąta z zablokowaniem kanałów RGB za pomocą ustawienia `D3DRS_COLORWRITEENABLE`.
4. Metoda `DrawHeatEntities` sceny renderuje encje ciepła do kanału alfa tekstury pomocniczej. W kanale tym znajduje się więc intensywność „falowania powietrza” w danym miejscu.
5. Podczas przerysowania tekstury pomocniczej do bufora ramki, `PpShader` przesuwa współrzędne próbkowania tej tekstury względem oryginalnych tak, aby ob-

raz był zniekształcony. Stopień tego przesunięcia dobiera na podstawie kanału alfa tekstury pomocniczej, a kierunkiem przesunięcia sterują wektory zapisane w kanałach RG specjalnej tekstury *Perturbation Map*, pokazanej na rys. 3.12.



Rys. 3.12. Tekstura narzędziowa *Perturbation Map*, w której jako kanały RG zapisane są wektory sterujące kierunkiem przesunięcia współrzędnych tekstury do próbkowania obrazu.

## 3.7. Renderowanie terenu

W przypadku scen typu otwartego, teren, czyli pofałdowana powierzchnia modelująca trawę, piasek itd., stanowi podłoże dla całego wirtualnego świata i wymaga specjalnego potraktowania. Zbudowany jest z regularnej siatki, której wierzchołki ułożone są w równych odstępach w płaszczyźnie $XZ$, a ich wysokość $y$ jest zależna od tzw. mapy wysokości (ang. *Heightmap*). Mapa wysokości to specjalna tekstura w odcieniach szarości, w której jasność pikseli wyznacza wysokość (patrz rys. 3.13).

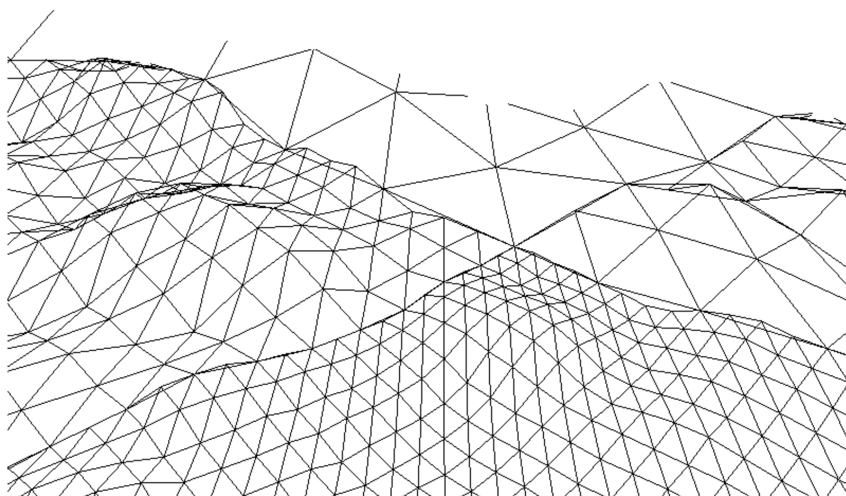


Rys. 3.13. Przykładowa mapa wysokości terenu.

Ponieważ generowanie siatki terenu na podstawie mapy wysokości jest procesem stosunkowo kosztownym obliczeniowo, wyniki są zapamiętywane do ponownego użycia w pliku binarnym, dzięki czemu siatka generowana jest tylko za pierwszym razem.

Kamera obejmuje swoim zasięgiem duży obszar terenu, z czego większość to fragmenty odległe. Dlatego zachodzi potrzeba nie tylko wybierania tych fragmentów terenu, które są widoczne w obszarze kamery, ale również dostosowania ich poziomu

szczegółowości do odległości od kamery. W tym celu zastosowany został *Geomipmapping* [39]. Ta prosta i skuteczna technika polega na podziale siatki terenu na kwadratowe sektory. Każdy sektor posiada kilka poziomów szczegółowości. W praktyce wystarczają odpowiednio skonstruowane bufory indeksów adresujące ten sam bufor wierzchołków. Ponadto jeden bufor indeksów może służyć do renderowania dowolnych sektorów terenu. Poziom szczegółowości danego sektora jest wybierany automatycznie na podstawie odległości tego sektora od kamery. Przykład pokazuje rys. 3.14.



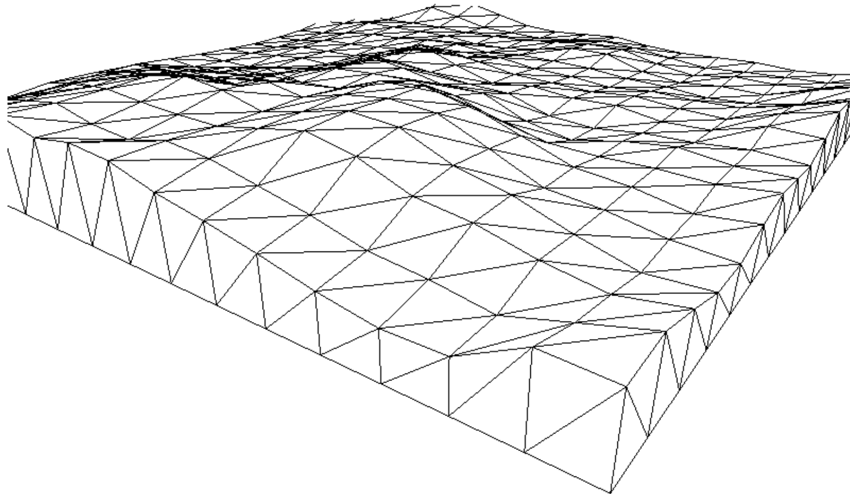
Rys. 3.14. Poziom szczegółowości sektorów terenu zostaje automatycznie dobrany zależnie od odległości sektora od kamery.

Na łączniach między sektorami o różnych poziomach szczegółowości mogą powstać „pęknięcia" (ang. *Cracks*). Aby im zapobiec, wykorzystana została prosta technika opisana w [40]. Polega ona na dodaniu do geometrii sektora dodatkowej „spódniczki" (ang. *Skirt*), która zapełnia ewentualne pęknięcia teksturą minimalizując niepożądany efekt. Pełną siatkę pojedynczego sektora terenu pokazuje rys. 3.15.

Sektory terenu nie są zapamiętane w żadnej strukturze drzewiastej takiej jak drzewo czwórkowe, ale stanowią po prostu dwuwymiarową tablicę. Sektory widoczne w zasięgu kamery są wybierane na podstawie testu przecięcia frustuma z AABB otaczającym dany sektor. Dzięki niemu odrzucenie sektora może się odbywać nie tylko w płaszczyźnie $XZ$, ale i w osi $Y$, jeśli na przykład kamera jest zwrócona do góry.

Oprócz pozycji wierzchołków, siatka terenu potrzebuje także wektorów normalnych używanych w oświetleniu i danych do teksturowania. Wektory normalne można policzyć zoptymalizowanym wzorem przedstawionym w [40], który uwzględnia pozycje 4 wierzchołków otaczających dany wierzchołek. Autor opracował jednak własny algorytm, który wymaga znacznie więcej obliczeń, ale w zamian daje dużo lepsze rezultaty, ponieważ uwzględnia wszystkie 8 wierzchołków dookoła danego. Kod tej procedury zawarty jest w metodzie `Terrain_pimpl::CalcNormals`.

Teren nie może być zazwyczaj pokryty pojedynczą teksturą. Zachodzi potrzeba



Rys. 3.15. Pojedynczy sektor terenu wraz z dodatkową geometrią zapobiegającą szczelinom na łączeniach między sektorami.

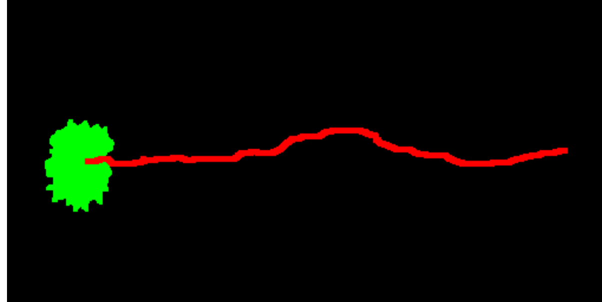
wybierania różnych „form terenu" — trawy, piasku, śniegu, ziemi itp. oraz płynnego przechodzenia między nimi. Popularną metodą stosowaną w tym celu jest *Texture Splatting* opisany w [43]. Autor opracował jednak własne rozwiązanie, w którym zamiast na osobnej teksturze, dane na temat formy terenu są zapisane w strukturze wierzchołka.

Na opis form terenu składają się dwa pliki. Pierwszy to plik tekstowy w specjalnym formacie, którego przykład pokazuje poniższy listing:

```
TerrainForms 1
0x000000 complex {
  0 {
    "Sand"
    TexScale = 0.234
  }
  50 {
    "Grass"
    TexScale = 0.345
  }
  162 {
    "Snow"
    TexScale = 0.294
  }
}
0xFF0000 simple {
  "Path"
  TexScale = 0.678
}
0x00FF00 simple {
  "Creep"
  TexScale = 0.938
}
```

Drugi plik to mapa form terenu zapisana w specjalnej teksturze, której przykład pokazuje rys. 3.16. Analizując obydwa te pliki można zauważyć, że kolor czerwony (`0xFF0000`) wyznacza „ścieżkę" (miejsca rysowane teksturą `"Path"` ze skalowaniem 0.678), a kolor zielony (`0x00FF00`) wyznacza miejsca rysowane teksturą `"Creep"`. Ko-

lor czarny (0x000000), pokrywający większość tekstury, jest zdefiniowany w pliku tekstowym jako forma złożona, w której wybór konkretnej tekstury jest zależny od wysokości. Obszary powyżej wysokości 162 (wysokość terenu jest w zakresie 0...255) to śnieg, obszary powyżej 50 to trawa, a niższe to piasek.



Rys. 3.16. Przykładowa mapa form terenu. Kolory wyznaczają poszczególne formy terenu.

Na podstawie tych dwóch plików obliczone zostają formy terenu używane przez każdy z sektorów. Na jednym sektorze używane mogą być co najwyżej 4 formy terenu. Do wierzchołków siatki terenu, których struktura jest pokazana poniżej, wpisane zostają (jako kanały ARGB elementu Diffuse) wagi blendingu poszczególnych form terenu w danym miejscu.

```
struct VERTEX
{
  VEC3 Pos;       // Semantyka D3DFVF_XYZ
  VEC3 Normal;    // Semantyka D3DFVF_NORMAL
  COLOR Diffuse;  // Semantyka D3DFVF_DIFFUSE
};
```

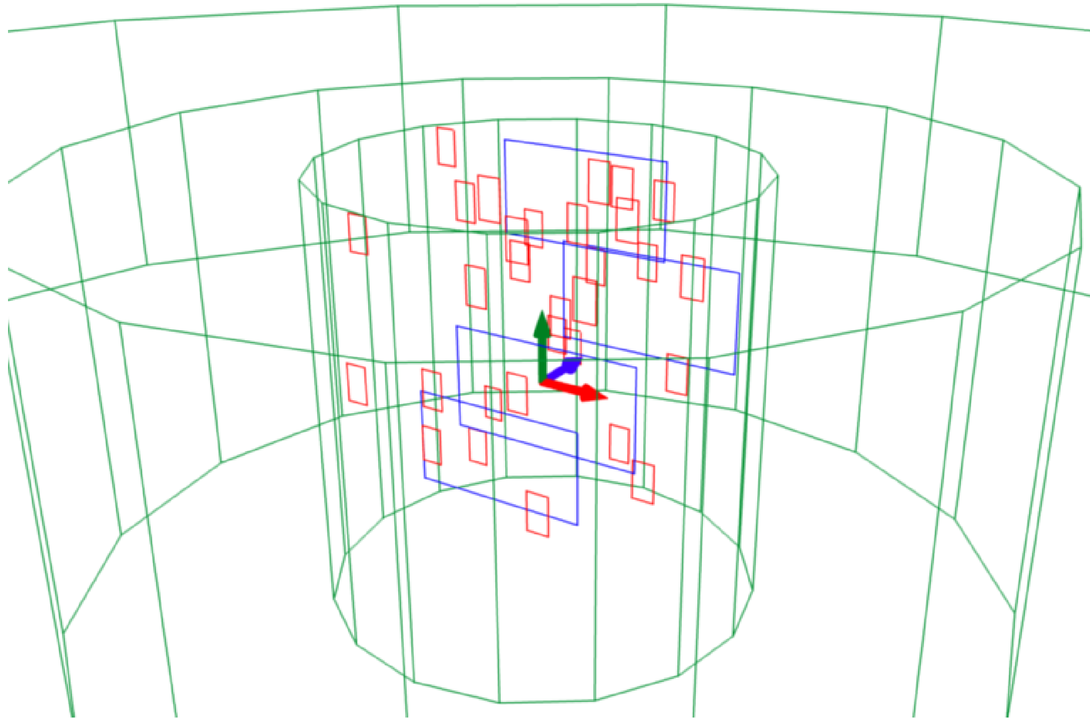
Następnie podczas rysowania danego sektora terenu, MainShader z włączonym makrem TERRAIN wykonuje samplowanie wszystkich 4 tekstur, a następnie ich mieszanie zależnie od ich wag w danym wierzchołku:

```
////// VERTEX SHADER
Out.TerrainTextureWeights = In.TerrainTextureWeights;
(...)
half2 Tex = half2(In.Pos.x, In.Pos.z);
Out.TerrainTex01.xy = Tex * TerrainTexScale.x;
Out.TerrainTex01.zw = Tex * TerrainTexScale.y;
Out.TerrainTex23.xy = Tex * TerrainTexScale.z;
Out.TerrainTex23.zw = Tex * TerrainTexScale.w;

////// PIXEL SHADER
MyDiffuseColor = tex2D(TerrainSampler0, In.TerrainTex01.xy);
MyDiffuseColor = lerp(MyDiffuseColor, tex2D(TerrainSampler1,
  In.TerrainTex01.zw), In.TerrainTextureWeights.a);
MyDiffuseColor = lerp(MyDiffuseColor, tex2D(TerrainSampler2,
  In.TerrainTex23.xy), In.TerrainTextureWeights.r);
MyDiffuseColor = lerp(MyDiffuseColor, tex2D(TerrainSampler3,
  In.TerrainTex23.zw), In.TerrainTextureWeights.g);
Out = MyDiffuseColor;
```

## 3.8. Efekt opadów atmosferycznych

Opady atmosferyczne takie jak deszcz i śnieg to zjawisko trudne do realistycznego zamodelowania. Autor wymyślił w tym celu własny sposób renderowania, który składa się z trzech elementów. Budowę efektu pokazuje rys. 3.17.

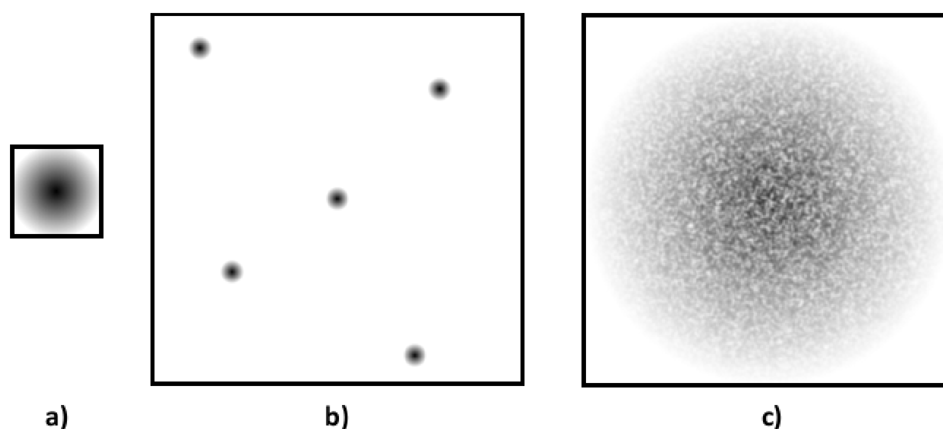


Rys. 3.17. Budowa efektu opadów atmosferycznych.

Na scenie typu otwartego kamera pokazuje rozległy obszar i rysowanie każdej cząstki spadającego deszczu czy śniegu nie jest dobrym pomysłem, bo musiałoby ich być bardzo dużo. Z drugiej strony, rysowanie płaszczyzny pokrytej przewijaną teksturą przedstawiającą „ścianę deszczu” tuż przed kamerą mogłoby być zbyt zauważalne i wyglądać nienajlepiej.

Dlatego autor połączył obydwa te podejścia. Prostokąty, oznaczone na wspomnianym rysunku kolorem czerwonym, są pokryte teksturą prezentującą pojedyncze krople deszczu albo płatki śniegu, pokazane na rys. 3.18 a). W ten sposób rysowane są opady w pobliżu kamery. Dalej znajduje się kilka obręczy (pokazanych na rys. 3.17 w kolorze zielonym), pokrytych teksturą prezentującą całą grupę cząstek (pokazaną na rys. 3.18 b). Symulują one opady w dalszej odległości od kamery. Trzecim składnikiem efektu są dodatkowe, duże prostokąty rysowane w pobliżu kamery (kolor niebieski na rys. 3.17) i pokryte teksturą szumu (rys. 3.18 c), które są symulacją „zamieci” — bardzo intensywnych opadów.

Tworzenie efektu opadów atmosferycznych, reprezentowanego przez klasę `Fall`, odbywa się z podaniem wypełnionej struktury `FALL_EFFECT_DESC`. Jej pola opisują dany efekt i nie podlegają zmianie w czasie jego działania. Częścią pierwszą efektu




Rys. 3.18. Tekstury używane podczas rysowania opadów atmosferycznych: a) pojedyncza cząsteczka śniegu, b) grupa cząsteczek śniegu, c) szum do „zamieci".

(pojedyncze cząsteczki) sterują pola: `ParticleTextureName` określające teksturę cząsteczki, `ParticleHalfSize` określające szerokość i wysokość cząsteczki, `UseRealUpDir` określające sposób zwracania prostokątów cząsteczek przodem do kamery, `MovementVec1` i `MovementVec2` wyznaczające zakres kierunków ruchu cząsteczek. Częścią drugą efektu (obręcze z przewijaną teksturą opadów) sterują pola: `PlaneTextureName` określające teksturę grupy cząsteczek, `PlaneTexScale` określające skalowanie współrzędnych tej tekstury. Ponadto określić trzeba kolor, przez który mnożone mają być próbki z tekstur. Tekstury są bowiem białe (na rysunkach tutaj przedstawione są w negatywie).

Podczas trwania efektu sterować można jego ogólną intensywnością podając liczbę 0...1. Klasa automatycznie przekłada ją na parametry poszczególnych składników efektu (liczba bliskich cząstek, liczba dalszych obręczy, liczba prostokątów „zamieci" itp.). Ponadto klasa uwzględnia pobrany ze sceny wektor wiatru, aby przedstawić opady stosownie do jego kierunku i intensywności. Zostało to zrealizowane za pomocą zaaplikowania do całej rysowanej przez tą klasę geometrii dodatkowego przekształcenia ścinania (ang. *Shearing*).

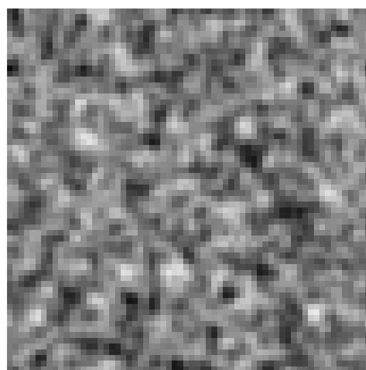
Dzięki możliwości określenia wielu parametrów i podania dowolnych tekstur, efekt opadów atmosferycznych jest bardzo elastyczny i daje duże możliwości. W demie dołączonym do pracy został wykorzystany do otrzymania deszczu i śniegu, ale równie dobrze może posłużyć do przedstawienia burzy piaskowej na pustyni, pyłków lecących od podłoża do góry, jakiś magicznych cząstek latających w różne strony itd.

## 3.9. Renderowanie trawy

Renderowanie trawy jest wyzwaniem, gdyż wymaga bardzo dużej ilości geometrii. Mimo że poszczególne źdźbła trawy nie są modelowane za pomocą trójkątów, ale ich obraz powstaje poprzez rysowanie równoległoboków przedstawiających teksturę całej kępki trawy, potrzebne jest wiele wierzchołków. Dlatego trawa rysowana jest tylko

w pobliżu kamery i płynnie zanika wraz z odległością.

Ponieważ trawa nie używa alfa–blendingu, tylko testu Alfa, zanikanie zostało zrealizowane za pomocą techniki opisanej w [47]. Wartość Alfa odczytana z tekstury trawy jest modyfikowana przez jasność pobraną z tekstury przedstawiającej rozmyty szum, pokazanej na rys. 3.19, odpowiednio przesuniętą zależnie od odległości do kamery. W ten sposób, im dalej dany trójkąt znajduje się od kamery, tym większy procent losowych pikseli jest niewidoczny. Intuicyjnie mogłoby się wydawać, że taka technika będzie wyglądała brzydko, że quady z trawą będą „powygryzane”. W praktyce jednak sprawdza się bardzo dobrze, bo quadów jest wiele i uwaga odbiorcy skupia się na ogólnym wrażeniu, nie na poszczególnych fragmentach trawy.



Rys. 3.19. Tekstura narzędziowa przedstawiająca rozmyty szum, używana do realizacji zanikania trawy z odległością.

Struktura wierzchołka trawy wygląda następująco:

```
struct VERTEX {
  VEC3 Pos;
  COLOR Diffuse;
  VEC2 Tex;
};
```

Składowe ARGB elementu `Diffuse` zostały wykorzystane do zapisania dodatkowych danych: `R` i `G` oznaczają rozsunięcie wierzchołka od środka quada odpowiednio w osi poziomej (w kierunku wektora „w prawo” kamery) i w osi pionowej (w kierunku wektora „do góry” kamery). Są przekształcane do zakresu $-2 \ldots 2$. `B` i `A` oznaczają losowe współczynniki wpływu odpowiednio pierwszego i drugiego wektora wiatru na dany wierzchołek. Są przekształcane do zakresu $-1 \ldots 1$. Współrzędne tekstury `Tex` są losowo odwracane w osi pionowej, aby jedne fragmenty trawy były lustrzanym odbiciem innych. To wprowadza dodatkową różnorodność.

Quady przedstawiające fragmenty trawy są rysowane w podobny sposób, jak cząsteczki w efekcie cząsteczkowym. Każdemu quadowi odpowiadają 4 wierzchołki i 6 indeksów, tworząc 2 trójkąty. Wszystkie 4 wierzchołki mają zapisaną tą samą pozycję — pozycję środka quada — i są od niej rozsuwane w kierunku wektorów „w prawo” i „do góry” pobranych z kamery. Dodatkowo niektóre wierzchołki (górna krawędź quada) są przesuwane w kierunku „w prawo” przez współczynnik służący do otrzymania efektu

falowania trawy na wietrze. Aby nie wszystkie fragmenty trawy poruszały się w tym samym kierunku, do shadera są przekazywane dwa współczynniki wiatru i w każdym wierzchołku ich wpływ jest ważony przez komponenty `B` i `A` elementu `Diffuse`. Oto kod shadera używanego do renderowania trawy:

```
void GrassVS(VS_INPUT In, out VS_OUTPUT Out)
{
  float2 QuadBias = In.Diffuse.rg * 4 - 2;
  float2 Randoms = In.Diffuse.ba * 2 - 1;
  float Wind = dot(WindFactors, Randoms);
  float3 WorldPos = In.Pos + (QuadBias.x + Wind) * RightDir +
    QuadBias.y * UpDir;
  Out.Pos = mul(float4(WorldPos, 1), ViewProj);
  Out.Tex = In.Tex;
  Out.NoiseTex.xy = float2(WorldPos.x + WorldPos.z, WorldPos.y);
  Out.NoiseTex.z = Out.Pos.z;
}

void GrassPS(VS_OUTPUT In, out half4 Out : COLOR0)
{
  Out = tex2D(Sampler, In.Tex) * Color;
  float Pos_z = In.NoiseTex.z;
  float MyFadeFactor = Pos_z * FadeFactors.x + FadeFactors.y;
  float Fade = tex2D(NoiseSampler, In.NoiseTex) + MyFadeFactor;
  Out.a *= saturate(Fade);
}
```

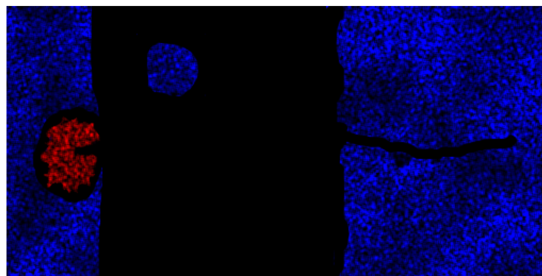
Gatunki trawy opisuje plik tekstowy w specjalnym formacie, podawany przy tworzeniu obiektu klasy `Grass` jako `GrassDescFileName`. Oto przykładowa treść:

```
GrassDesc
Texture {
  "GrassTexture"
  512, 128
}
Grasses {
  "Grass" {
    CX = 0.7, 0.1
    CY = 0.5, 0.05
    Tex = 0, 0, 207, 127
    Wind = 1
  }
    "Flower" {
    CX = 0.5, 0.1
    CY = 0.5, 0.05
    Tex = 257, 0, 397, 127
    Wind = 0.8
  }
  "Stone" {
    CX = 0.1, 0.05
    CY = 0.05, 0.03
    Tex = 415, 39, 483, 72
    Wind = 0
  }
}
DensityMap {
  R {
    "Stone" 5
  }
  G {
    "Grass" 5,
  }
```

```
  B {
    "Grass" 5,
    "Flower" 2
  }
}
```

Wszystkie rodzaje obiektów muszą występować na wspólnej teksturze. Powyższy przykład definiuje 3 takie rodzaje: trawa, kwiat i kamień. Są one wyznaczone przez parametry: `CX`, `CY` — szerokość i wysokość quada, we współrzędnych świata, `Tex` — prostokąt wyznaczający obraz danego rodzaju obiektu na teksturze w pikselach, `Wind` — stopień wpływu wiatru (kamień w ogóle nie kołysze się na wietrze).

Ostatnia sekcja pliku tekstowego definiuje znaczenie kanałów RGB na specjalnej teksturze podawanej podczas tworzenia efektu jako `DensityMapFileName`. Tekstura ta wyznacza miejsca na terenie, które mają być pokryte poszczególnymi rodzajami trawy oraz jej intensywność. Przykład takiej tekstury pokazuje rys. 3.20. Na przykład na fragmencie terenu, któremu na tej teksturze odpowiada piksel o intensywności składowej niebieskiej 255, powstanie 5 quadów trawy i 2 quady kwiatów, natomiast tam, gdzie piksel ma kolor czerwony o intensywności 100, powstaną 2 kamienie. Poszczególne quady są rozlokowane w losowych miejscach w ramach danego fragmentu terenu wyznaczanego przez sąsiednie wierzchołki mapy wysokości.



Rys. 3.20. Przykładowa tekstura opisująca rozmieszczenie rodzajów trawy na terenie.

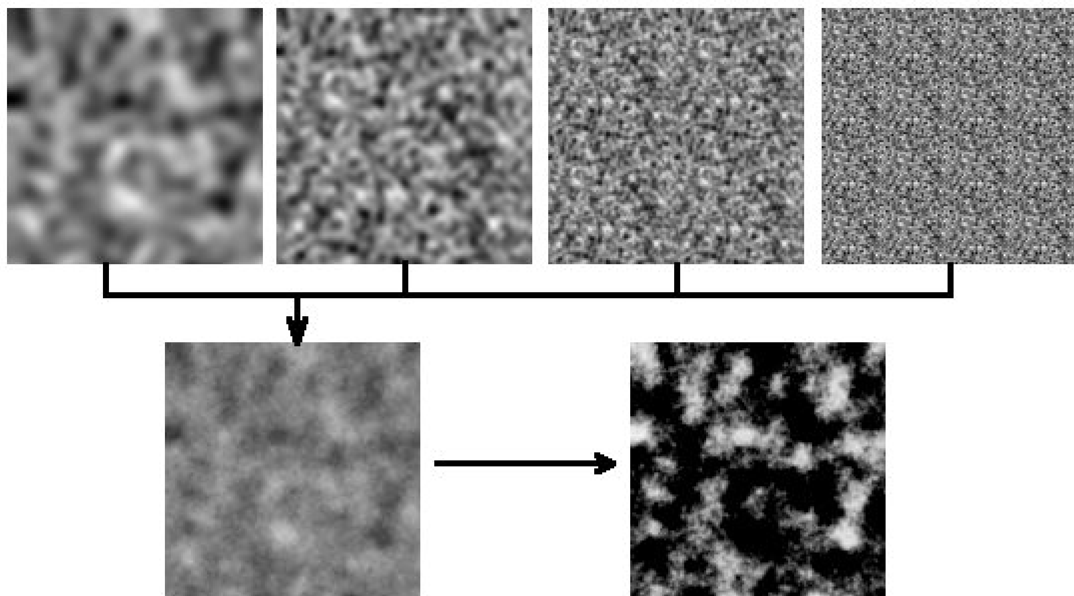
Ponieważ generowanie geometrii trawy jest kosztowne obliczeniowo, wygenerowana geometria dla ostatnio wyświetlanych sektorów terenu jest spamiętywana do ponownego użycia.

## 3.10. Renderowanie nieba

Niebo reprezentowane przez klasę `ComplexSky` jest renderowane w kilku etapach. Jego statyczne parametry opisuje plik tekstowy w specjalnym formacie. Pierwszy składnik nieba to tło, rysowane jako hemisfera. Jest ona pokryta gradientem — płynnym przejściem od koloru horyzontu do koloru zenitu. Dodatkowo w nocy przedstawiana jest na niej tekstura gwiazd. Datą i czasem steruje parametr `Time`, w którym jedna jednostka to jedna doba. $0$ oznacza północ, $0.5$ południe, $1$ północ następnego dnia itd. Kolorami horyzontu i zenitu, interpolowanymi zależnie od pory dnia, steruje część pliku z opisem nieba taka jak w przykładzie:

```
Background {
  0.0 0x282B24 0x060606, // noc - czarno
  0.3 0x282B24 0x060606, // noc - czarno
  0.4 0xB8454A 0x5783C2, // rano - wschód słońca
  0.5 0x7f91a7 0x99CCF9, // południe - jasno
  0.8 0xD0D3FE 0x101F5A, // popołudnie - błękit
  0.9 0xED712B 0x001C43  // wieczór - zachód słońca
}
```

Drugi składnik nieba to chmury generowane proceduralnie za pomocą szumu Perlina [49]. Przybliżenie szumu Perlina generowanego w czasie rzeczywistym na GPU zostało zrealizowane poprzez czterokrotne próbkowanie tekstury zwykłego szumu. Każde próbkowanie odbywa się z odpowiednio przeskalowanymi współrzędnymi tekstury, a ich wyniki są sumowane z wagami tak, aby odpowiadać oktawom szumu Perlina. Następnie wynik podlega dodatkowym przekształceniom, m.in. funkcją wykładniczą. Chmura jest widoczna tylko w tych miejscach, w których wynik końcowy ma wartość powyżej określonej granicy. Granicę tą, jak również „ostrość" chmur można regulować w czasie działania, co pozwala płynnie zmieniać chmury np. stosownie do ogólnie pojętej pogody, wpływającej również na jasność oświetlenia czy intensywność opadów. Dodatkowo, kolor chmury jest interpolowany między dwoma podanymi kolorami zależnie od tej wartości intensywności w danym miejscu, co w pewnym stopniu imituje wrażenie przestrzenności chmur. Cały ten proces zilustrowany został na rys. 3.21.



Rys. 3.21. Proces generowania chmur jako szum Perlina wyliczany w czasie rzeczywistym na GPU.

Trzecim składnikiem nieba są ciała niebieskie — np. Słońce, Księżyc, duże gwiazdy czy planety. Są one opisywane przez strukturę `CELESTIAL_OBJECT_DESC`. Każde ciało niebieskie jest rysowane jako kwadrat pokryty wybraną teksturą. Jego ruch opisuje orbita, po jakiej obraca się wokół kamery. Podany musi zostać wektor normalny tej orbity, a także okres i faza obiegu. Ponadto istnieje możliwość modyfikowania koloru,

w jakim rysowana jest tekstura ciała niebieskiego (wraz z przezroczystością w kanale Alfa) zależnie od pory dnia i/lub od wysokości na niebie. Klasa renderująca niebo wieloetapowo buforuje pośrednie wyniki obliczeń, aktualizując je tylko przy pierwszym użyciu po zmianie danych wejściowych. Obliczanie pozycji wierzchołków quada ciała niebieskiego można przedstawić w pseudokodzie następująco (`Time` to aktualny czas, `Desc` to struktura opisująca orbitę po której krąży ciało niebieskie):

```
// ComplexSky_pimpl::EnsureCelestialObjectDir
vec3 RightDir = normalize(cross(vec3(0, 1, 0), Desc.OrbitNormal))
vec3 UpDir = normalize(cross(Desc.OrbitNormal, RightDir))
float Angle = Time * (2*PI / Desc.Period) + Desc.Phase
vec3 Dir = cos(Angle) * RightDir + sin(Angle) * UpDir

// ComplexSky_pimpl::FillQuadVertices
vec3 Forward = Dir
vec3 Right = Desc.OrbitNormal
vec3 Up = cross(Forward, Right)
matrix RotMat = AxesToMatrix(Right, Up, Forward)
float Delta = SKYDOME_RADIUS * tan(Desc.Size * 0.5)
Vertices[0] = vec3(-Delta, -Delta, SKYDOME_RADIUS)
Vertices[1] = vec3(-Delta,  Delta, SKYDOME_RADIUS)
Vertices[2] = vec3( Delta,  Delta, SKYDOME_RADIUS)
Vertices[3] = vec3( Delta, -Delta, SKYDOME_RADIUS)
Vertices[0] = Transform(Vertices[0], RotMat);
Vertices[1] = Transform(Vertices[1], RotMat);
Vertices[2] = Transform(Vertices[2], RotMat);
Vertices[3] = Transform(Vertices[3], RotMat);
```

## 3.11. Renderowanie drzew

Drzewa są przez silnik renderowane w specjalny sposób, inaczej niż zwyczajne modele. Kształt pnia, korzeni i gałęzi jest generowany proceduralnie. Autor opracował własny algorytm tego generowania, inspirowany [44], jednak znacznie prostszy. Pień, korzenie i gałęzie poziomu pierwszego mają siatkę o kształcie cylindrycznym, natomiast mniejsze gałęzie są zbudowane z dwóch krzyżujących się czworokątów. Gałęzie powstają rekurencyjnie, a każdy poziom opisuje następująca struktura:

```
struct TREE_LEVEL_DESC {
  bool Visible;
  uint SubbranchCount;
  float SubbranchRangeMin, SubbranchRangeMax;
  float SubbranchAngle, SubbranchAngleV;
  float Length, LengthV;
  float LengthToParent;
  float Radius, RadiusV;
  float RadiusEnd;
  float LeafCount, LeafCountV;
  float LeafRangeMin, LeafRangeMax;
};
```

Pole `SubbranchCount` to liczba odgałęzień. Pola `SubbranchRangeMin` i `SubbranchRangeMax` wyznaczają procentowo zakres minimalnej i maksymalnej długości gałęzi, na której mogą pjawiać się podgałęzie. Kąt nachylenia odgałęzień względem kierunku danej gałęzi jest liczbą losową o wartości `SubbranchAngle±`

SubbranchAngleV. Długość gałęzi danego poziomu jest liczbą losową o wartości Length±LengthV. Pole LengthToParent to dodatkowy składnik długości wyrażony w procentach długości gałęzi nadrzędnej. Grubość gałęzi (promień) wynosi Radius±RadiusV. RadiusEnd to promień zakończenia gałęzi, wyrażony w procentach promienia początkowego. Liczba liści osadzonych na gałęzi danego poziomu jest wyrażona przez LeafCount±LeafCountV, a zakres procentowy długości gałęzi, na jakiej mogą się pojawiać liście, to
LeafRangeMin...LeafRangeMax.

Sposób renderowania liści wzorowany jest na obserwacji działania technologii SpeedTree [45]. Tekstura przedstawiająca grupę liści jest rysowana jako prostokąt zwrócony zawsze przodem do kamery, o środku w punkcie leżącym w określonym miejscu na określonej gałęzi. Struktura wierzchołka używanego do renderowania drzew (zarówno pień, gałęzie, jak i liście są renderowane razem) wygląda następująco:

```
struct TREE_VERTEX {
  VEC3 Pos;      // Semantyka D3DFVF_XYZ
  VEC3 Normal;   // Semantyka D3DFVF_NORMAL
  COLOR Diffuse; // Semantyka D3DFVF_DIFFUSE
  VEC2 Tex;      // Semantyka D3DFVF_TEXCOORDSIZE2(0)
};
```

Wykorzystanie składowych ARGB elementu Diffuse jest takie samo, jak w wierzchołku używanym do renderowania trawy (p. 3.9). Samo renderowanie też odbywa się w podobny sposób jak w przypadku trawy. Wierzchołki należące do liści są rozsuwane w kierunku „w prawo” i „do góry” pobranym z kamery tworząc czworobok zwrócony zawsze przodem do kamery. Ponadto są one przesuwane przez sumę ważoną dwóch współczynników wiatru, co daje wrażenie falowania liści na wietrze.

Oświetlenie drzew również realizowane jest w sposób nietypowy. Jest ono obliczane zawsze metodą *Half–Lambert*, dzięki czemu drzewo widziane od strony przeciwnej niż źródło światła nie jest całkowicie czarne. Jako wektor normalny wierzchołków należacych do grup liści wpisany jest wektor wskazujący kierunek od środka drzewa do danej grupy liści. Dzięki temu grupy leżace w koronie drzewa po tej stronie, od której pada światło, są silniej oświetlone.

Zrzut ekranu z tymczasowego edytora drzew, pokazujący przykładowy kształt gałęzi przed dodaniem liści, pokazuje rys. 3.22.

## 3.12. Renderowanie wody

Powierzchnia wody renderowana jest w przedstawionym silniku jako płaszczyzna. Dlatego właściwie cały algorytm wizualizacji wody zawiera przeznaczony do tego celu Pixel Shader. Jego kod HLSL przedstawia poniższy listing:




Rys. 3.22. Tymczasowy edytor drzew. Przykładowy kształt gałęzi przed dodaniem liści.

```
float3 DirToCam = normalize(In.VecToCam);
DirToCam.y = abs(DirToCam.y);

float3 NormalSample1 =
  tex2D(NormalSampler, In.NormalTex1).rgb * 2 - 1;
float3 NormalSample2 =
  tex2D(NormalSampler, In.NormalTex2).rgb * 2 - 1;
float3 FinalNormal =
  normalize(((NormalSample1 + NormalSample2)/2).xzy);

float3 ReflectedLight = reflect(-DirToLight, FinalNormal);
float3 ReflectedCam = reflect(-DirToCam, FinalNormal);

float EnvFactor = ReflectedCam.y;
float3 EnvColor = lerp(HorizonColor, ZenithColor, EnvFactor);

float3 MyWaterColor = WaterColor;
MyWaterColor +=
  tex2D(CausticsSampler, In.CausticsTex) * CausticsColor;

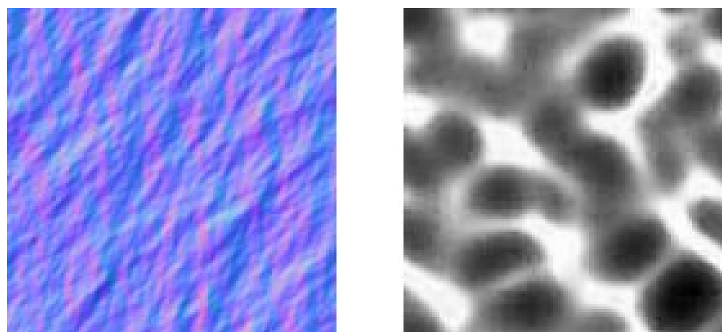
float FresnelFactor = 1 - saturate(dot(FinalNormal, DirToCam));
float3 FresnelColor = lerp(MyWaterColor, EnvColor, FresnelFactor);

float SpecularFactor = pow(saturate(dot(ReflectedLight, DirToCam)),
  SpecularExponent);
float3 SpecularColor = LightColor * SpecularFactor;

Out.rgb = FresnelColor + SpecularColor;
Out.a = lerp(WaterColor.a, 1, max(FresnelFactor, SpecularFactor));
```

Wyliczenie koloru powierzchni wody w danym miejscu składa się z kilku etapów. Pierwszym jest otrzymanie wektora normalnego, który symuluje pofalowaną powierzchnię wody. Powstaje on przez uśrednienie próbki z dwóch kopii mapy normalnych (patrz rys. 3.23 po lewej), przesuwanych w różnych kierunkach. Wektor `FinalNormal` jest wyrażony w układzie modelu.

Następnie wyliczony zostaje wektor kierunku patrzenia kamery odbity od powierzchni wody — `ReflectedCam`. Zostaje on wykorzystany do określenia koloru nieba `EnvColor`, jaki odbija się w wodzie w danym miejscu. Zastosowana w tym celu inter-



Rys. 3.23. Tekstury używane podczas renderowania wody. Po lewej: mapa normalnych. Po prawej: kaustyki.

polacja od koloru horyzontu do koloru zenitu wg składowej $y$ tego wektora jest prostym przybliżeniem mapowania środowiskowego, które mogłoby być w tym miejscu użyte, aby otrzymać prawdziwe odbicie w wodzie obrazu otaczającej ją sceny.

Wartość `MyWaterColor` powstaje najpierw przez przypisanie stałej przechowującej kolor własny wody. Wbrew intuicji jest to kolor zielony, gdyż barwa niebieska pochodzi od odbicia nieba opisanego wyżej. Następnie dodawana jest do tej wartości próbka tekstury kaustyk (pokazanej na rys. 3.23 po prawej), pomnożona przez kolor kaustyk. Teoretycznie tekstura ta powinna być mapowana na dnie zbiornika, jednak dla uproszczenia jest adresowana jak na powierzchni wody.

Wyliczany dalej współczynnik `FresnelFactor` przyjmuje wartości od 0, kiedy kamera patrzy prostopadle od góry na powierzchnię wody, do 1, kiedy kierunek patrzenia kamery jest równoległy do powierzchni wody w danym miejscu. Teoretycznie powinien on zostać podniesiony do potęgi, ale empirycznie stwierdzone zostało, że najlepszy wizualnie efekt daje wykładnik równy 1. Na podstawie tego współczynnika obliczony zostaje końcowy kolor wody `FresnelColor` jako interpolacja między kolorem własnym wody a kolorem nieba odbitego od jej powierzchni.

Dalej obliczony zostaje metodą Phonga [14] odblask od źródła światła — `SpecularColor`. Jest to najbardziej efektowny składnik całego efektu wody. Jego kolor zostaje dodany do koloru końcowego.

Stopień przezroczystości (kanał Alfa) również jest zależny od kąta nachylenia kierunku patrzenia kamery do powierzchni wody w danym miejscu. Dzięki temu, spoglądając na wodę od góry, użytkownik widzi dno, kaustyki i kolor wody (zielony). Kiedy natomiast spogląda na powierzchnię wody od boku, widzi wodę o barwie niebieskiej, pochodzącej od odbicia nieba. To daje stosunkowo realistyczny efekt wymagając przy tym niewielkiej złożoności obliczeniowej.

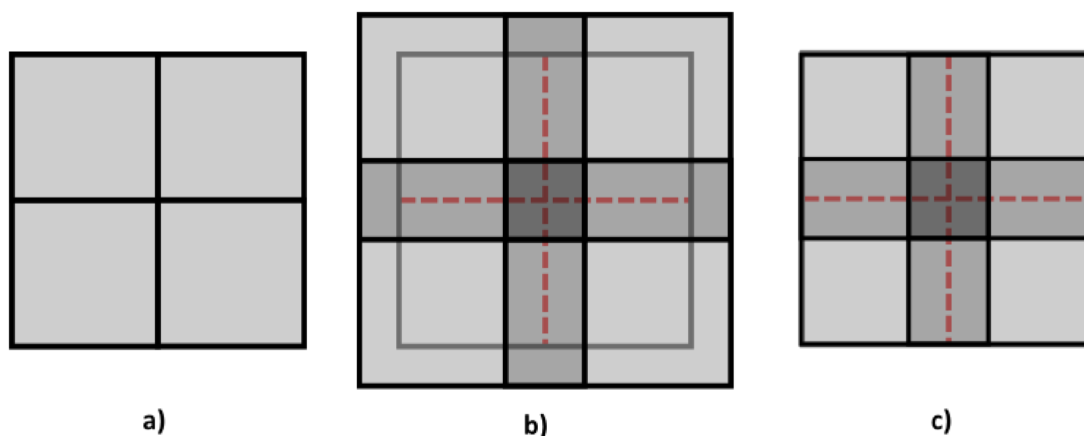
## 3.13. Implementacja swobodnego drzewa ósemkowego

Ponieważ silnik przeznaczony jest do pokazywania scen zarówno typu otwartego, jak i zamkniętego, jako technikę podziału przestrzeni autor wybrał rozwiązanie, które do-

brze sprawdza się w obydwu tych zastosowaniach — swobodne drzewo ósemkowe [11]. Polega ono na rekurencyjnym podziale prostopadłościanu otaczającego cały wirtualny świat na 8 mniejszych prostopadłościanów. Wskaźnik do każdej encji utworzonej na scenie znajduje się na liście encji w jednym z węzłów drzewa, zagnieżdżonym tak głęboko, jak to możliwe, o ile sfera otaczająca tą encję mieści się w całości w prostopadłościanie opisującym granice danego węzła drzewa. Hierarchia węzłów drzewa ósemkowego nie ma przy tym nic wspólnego z hierarchią, jaką mogą tworzyć encje, aby ich przekształcenia były składane. Poniższy listing przedstawia definicję struktury węzła drzewa (język C++):

```
struct ENTITY_OCTREE_NODE {
  BOX Bounds;
  ENTITY_OCTREE_NODE *Parent;
  ENTITY_OCTREE_NODE *SubNodes[8];
  ENTITY_VECTOR Entities;
};
```

Dokładny opis techniki drzew ósemkowych nie leży w zakresie niniejszej pracy. Zilustrowania wymaga jedynie modyfikacja kształtu prostopadłościanów określających węzły względem węzła nadrzędnego, jakiej autor dokonał w stosunku do algorytmu zaproponowanego w [11]. Opiera się ona na obserwacji, że obiekty znajdujące się w danym węźle drzewa na pewno zawierają się w całości w jego AABB, więc wykraczanie AABB węzłów podrzędnych poza obszar AABB węzła nadrzędnego nie ma sensu. Ilustruje to rys. 3.24.




Rys. 3.24. Kształt AABB podwęzłów drzewa ósemkowego względem AABB węzła nadrzędnego, w przekroju 2D. a) Zwykłe drzewo ósemkowe. b) Swobodne drzewo ósemkowe wg [11]. c) Swobodne drzewo ósemkowe w implementacji autora.

Drzewo reprezentuje klasa `EntityOctree`, będąca częścią implementacji wewnętrznej silnika, niewidocznej dla użytkownika. Scena używa jej do przechowywania encji. Ponieważ encje mogą być tworzone i usuwanie w czasie działania, a także zmieniać swoje parametry, zachodzi potrzeba dynamicznej reorganizacji drzewa — łączenia lub dzielenia węzłów. Metoda dodająca encję do drzewa to `AddEntityToNode`. Jej algorytm można przedstawić w pseudokodzie następująco:

```
"function AddEntityToNode(Węzeł, Encja)"
if (Węzeł nie jest liściem):
  foreach (Podwęzeł in Węzeł.Podwęzły):
    if (Encja.Sfera_otaczająca zawiera się w Podwęzeł.AABB):
      AddEntityToNode(Podwęzeł, Encja) // Rekurencja
      return
  Węzeł.Dodaj_encję(Encja)
else:
  Węzeł.Dodaj_encję(Encja)
  if (Węzeł.Liczba_encji > ENTITY_OCTREE_SPLIT_ENTITY_COUNT):
    SplitNode(Węzeł)
```

Metoda ta próbuje dodać encję do węzła tak głęboko zagnieżdżonego, jak to możliwe. Jeśli sfera otaczająca tą encję nie zawiera się w całości w prostopadłościanie otaczającym żadnego z podwęzłów, encja trafia do węzła aktualnego (encje mogą być zawarte nie tylko w liściach, ale w dowolnych węzłach drzewa). Jeśli dany węzeł nie posiada podwęzłów, ale po dodaniu do niego nowej encji liczba wszystkich encji na jego liście przekroczyła granicę określoną przez stałą, wywołana zostaje metoda dzieląca węzeł na podwęzły. Jej algorytm w pseudokodzie to:

```
"function SplitNode(Węzeł)"
box AABB[8] = BuildSubBounds(Węzeł.AABB)
for (i = 0..7):
  Węzeł.Podwęzły[i] = Inicjalizuj_węzeł(AABB[i])
foreach (Encja in Węzeł.Encje):
  foreach (Podwęzeł in Węzeł.Podwęzły):
    if (Encja.Sfera_otaczająca zawiera się w Podwęzeł.AABB):
      Podwęzeł.Encje.Dodaj(Encja)
      Węzeł.Encje.Usuń(Encja)
      break
```

Metoda ta tworzy i inicjalizuje 8 podwęzłów dla podanego węzła. Ich alokacja odbywa się z użyciem szybkiego alokatora FreeList (p. 2.3). Metoda `BuildSubBounds` użyta zostaje w celu wyliczenia parametrów prostopadłościanów otaczających podwęzły na podstawie przekazanego prosopadłościanu węzła. Dalej algorytm przechodzi kolekcję encji zapamiętanych w węźle i każdy z nich próbuje przenieść do pierwszego z podwęzłów, w którym sfera otaczająca encji mieści się w całości.

Reorganizacja drzewa potrzebna jest także podczas usuwania encji. Odpowiada za to metoda `RemoveEntity`. Jej algorytm w pseudokodzie przedstawia następujący listing:

```
"function RemoveEntity(Encja)"
Węzeł = Encja.Węzeł_w_którym_leży
Węzeł.Encje.Usuń(Encja)
if (Węzeł nie jest korzeniem):
  TryJoin(Węzeł.Węzeł_nadrzędny)
```

Każda encja ma zapamiętany węzeł drzewa ósemkowego, w którym leży. Na jego podstawie metoda lokalizuje encję na liście tego węzła i usuwa ją z tej listy. Ponieważ usunięcie encji może być okazją do połączenia węzła, w którym leżała, z jego rodzeństwem, wywołana zostaje metoda `TryJoin` dla węzła nadrzędnego.

```
"function TryJoin(Węzeł)"
uint LiczbaEncji = Węzeł.Encje.Liczba
foreach (Podwęzeł in Węzeł.Podwęzły):
  if (Podwęzeł nie jest liściem):
    return
  LiczbaEncji = LiczbaEncji + Podwęzeł.Encje.Liczba
if (LiczbaEncji < ENTITY_OCTREE_JOIN_ENTITY_COUNT):
  foreach (Podwęzeł in Węzeł.Podwęzły):
    foreach (Encja in Podwęzeł.Encje):
      Węzeł.Encje.Dodaj(Encja)
    Usuń Podwęzeł
```

Powyższa metoda najpierw zlicza liczbę encji zawartych w podanym węźle oraz w jego podwęzłach. Sprawdza przy okazji, czy wszystkie jego podwęzły są liśćmi. Jeśli nie są (mają swoje podwęzły), to podany węzeł nie nadaje się do połączenia i cały algorytm zostaje przerwany. Jeśli wynikowa liczba encji jest poniżej progu określonego stałą, algorytm łączy przekazany mu węzeł. W tym celu najpierw przepisuje wszystkie encje z list swoich podwęzłów do swojej listy, a następnie usuwa swoje podwęzły.

Trzecim przypadkiem jest sytuacja, kiedy w istniejącej encji zmieniają się parametry kształtu jej sfery otaczającej — tj. jej pozycja lub promień. Zostaje wówczas wywołana metoda drzewa ósemkowego `OnEntityParamsChange`. Oto jej algorytm w pseudokodzie:

```
"function OnEntityParamsChange(Encja)"
Węzeł = Encja.Węzeł_w_którym_leży
if (Encja.Sfera_otaczająca zawiera się w Węzeł.AABB):
  if (Węzeł nie jest liściem):
    foreach (Podwęzeł in Węzeł.Podwęzły):
      if (Encja.Sfera_otaczająca zawiera się w Podwęzeł.AABB):
        Węzeł.Encje.Usuń(Encja)
        AddEntityToNode(Podwęzeł, Encja)
        break
else:
  NowyWęzeł = Węzeł
  Pętla:
    if (NowyWęzeł jest korzeniem):
      break
    NowyWęzeł = NowyWęzeł.Węzeł_nadrzędny
    if (Encja.Sfera_otaczająca zawiera się w NowyWęzeł.AABB):
      break
  RemoveEntity(Encja)
  AddEntityToNode(NowyWęzeł, Encja)
```

Powyższa metoda korzysta z wcześniej przedstawionych, aby zaktualizować węzeł, w którym zawiera się zmieniona encja, a przy okazji wykonać wymaganą reorganizację drzewa. Rozpatruje ona dwa główne przypadki. Pierwszy to sytuacja, kiedy sfera otaczająca zmienioną encję nadal zawiera się w węźle, w którym encja dotychczas leżała. Wówczas jedyne co można zrobić to sprawdzić, czy po zmianie parametrów encja nie zaczęła mieścić się w całości w jednym z podwęzłów. Jeśli tak jest, zostaje ona usunięta ze swojego węzła i dodana do określonego podwęzła metodą `AddEntityToNode`. Drugi przypadek to sytuacja, kiedy zmieniona encja przestała mieścić się w AABB swojego węzła. Wówczas pętla przechodzi ścieżkę w górę drzewa poszukując najniższego z węzłów, w którym jej nowa sfera otaczająca się zawiera. Kiedy taki węzeł zostaje

znaleziony, encja jest usuwana z listy swojego starego węzła i dodawana do drzewa na poziomie znalezionego węzła metodą `AddEntityToNode`. Trzeba podkreślić, że metoda ta wykona rekurencyjne przejście w dół drzewa celem znalezienia najbardziej zagnieżdżonego węzła, do którego encja może trafić.

Korzystanie z drzewa polega na zadawaniu zapytań, których celem jest zwrócenie zbioru encji potencjalnie przecinających podany obiekt geometryczny. Dzięki strukturze hierarchicznej wyszukiwanie to odbywa się z szybkim eliminowaniem całych gałęzi (zawierających duże grupy encji). Można to porównać do logarytmicznego czasu, w jakim wykonuje się algorytm wyszukiwania binarnego w tablicy jednowymiarowej. Zależnie od sytuacji zachodzi potrzeba zadawania różnych zapytań. Służą do tego metody publiczne drzewa: `FindEntities_Frustum` pytająca o listę encji przecinających frustum, `FindEntities_SpotLight` pytająca o listę encji w zasięgu podanego światła latarki, `FindEntities_PointLight` pytająca o listę encji w zasięgu podanego światła punktowego, `FindEntities_DirectionalLight` pytająca o listę encji rzucających cień na obszar widoczny w kamerze w kierunku padania podanego światła kierunkowego oraz `RayCollision` sprawdzająca kolizję promienia z encjami. Poniższy listing przedstawia w pseudokodzie algorytm pierwszej z nich. Implementacja pozostałych wygląda podobnie.

```
"function FindEntities_Frustum(Frustum, out Encje)"
FindEntities_Frustum_Node(Korzeń, Frustum, Encje, false)

"function FindEntities_Frustum_Node(Węzeł, Frustum, out Encje,
  Wewnątrz)"
if (Węzeł nie jest liściem):
  foreach (Podwęzeł in Węzeł.Podwęzły):
    if (Wewnątrz):
      FindEntities_Frustum_Node(Podwęzeł, Frustum, Encje, true)
    else if (Podwęzeł.AABB zawiera się w Frustum):
      FindEntities_Frustum_Node(Podwęzeł, Frustum, Encje, true)
    else if (Podwęzeł.AABB przecina Frustum):
      FindEntities_Frustum_Node(Podwęzeł, Frustum, Encje, false)
foreach (Encja in Węzeł.Encje):
  if (Encja.Widoczna):
    if (Wewnątrz lub Encja.Sfera_otaczająca przecina Frustum):
      Encje.Dodaj(Encja)
```

Rekurencyjna metoda `FindEntities_Frustum_Node` składa się z dwóch etapów. Pierwszy to przechodzenie podwęzłów. Drugi to sprawdzenie encji zawartych w danym węźle. Na uwagę zasługuje tu parametr typu logicznego `Wewnątrz`, który służy do optymalizacji wydajności. Kiedy stwierdzone zostaje, że AABB pewnego węzła drzewa zawiera się w całości wewnątrz przekazanego frustuma, wówczas `Wewnątrz` przyjmuje wartość `true`. To sygnalizuje, że wszystkie podwęzły, jak również wszystkie encje leżące w danym węźle i jego podwęzłach na pewno przecinają frustum i nie trzeba ich już testować. Tylko kiedy AABB węzła przecina frustum, ale nie zawiera się w nim w całości, `Wewnątrz` ma wartość `false` i wtedy potrzebne jest testowanie przecięcia z frustumem sfer otaczających encje oraz AABB podwęzłów.

## 3.14. Budowa mapy

Mianem mapy w opisywanym silniku określana jest geometria przedstawiająca ściany, podłogi, sufity, korytarze, pomieszczenia, schody itp., wewnątrz których porusza się użytkownik w scenach typu zamkniętego. W tym sensie mapa przypomina siatki modeli, ale jednak nie może być traktowana w ten sam sposób.

W każdej chwili widoczny jest tylko niewielki fragment mapy. Dlatego zachodzi potrzeba zastosowania podziału przestrzeni, który pozwoli szybko odrzucać niewidoczne fragmenty geometrii. Autor ponownie zastosował w tym celu swobodne drzewo ósemkowe. Ponieważ mapa składa się z trójkątów, a ponadto pozostaje niezmienna przez cały czas działania programu, przechowywanie fragmentów mapy w jednym drzewie z encjami nie jest dobrym pomysłem. Dlatego mapa jest osobnym typem obiektu w silniku i posiada własny kod do realizacji wszystkich tych zadań.

Specjalna wtyczka do programu graficznego Blender napisana w języku Python eksportuje scenę do tekstowego formatu pośredniego z myślą o przetworzeniu na mapę. Wtyczka ta znajduje się w pliku `TFQ_QMAP_TMP_Exporter.py`. Narzędzie konsolowe Tools przetwarza plik w formacie QMSH.TMP do binarnego formatu docelowego QMAP. Format ten jest następnie wczytywany przez klasę `QMap` silnika, która udostępnia do odczytu dane. Wreszcie, dane te są wykorzystywane przez klasę `QMapRenderer` do wyświetlania mapy jako część sceny.

Autor postanowił nie opisywać dokładnie w niniejszej pracy formatu pliku QMAP, ponieważ nie jest to format ostateczny. W prawdziwych zastosowaniach mapa, oprócz geometrii ścian, podłóg itp., musi zawierać także dane na temat świateł i różnego rodzaju encji — ich pozycji i parametrów. Na przykład w przypadku gry z gatunku FPS mogłyby to być wrogie potwory, broń, paczki z amunicją, apteczki i przyciski otwierające drzwi. Wstawianie takich encji można sobie wyobrazić w programie graficznym typu Blender jako obiekty puste (typu *Empty*) lub specjalnie oznaczone, proste siatki jak prostopadłościan. Jednak lepiej byłoby używać w tym celu specjalnego edytora map takiego jak QuArK, DeleD lub własnego edytora napisanego specjalnie na potrzeby danej gry. Dane, jakie powinna przechowywać mapa, silnie zależą od konkretnego zastosowania, dlatego nie sposób przewidzieć ich pisząc ogólny silnik graficzny.

Geometria mapy zapisana jest w pliku i wczytywana do pamięci w formie swobodnego drzewa ósemkowego, a jego węzły są alokowane za pomocą szybkiego alokatora FreeList (p. 2.3) — tak jak ma to miejsce w przypadku drzewa encji opisanego w poprzednim podrozdziale. W mapie jednak węzły drzewa przechowują fragmenty geometrii, a więc trójkąty. Cała mapa występuje w dwóch kopiach, każda w osobnym drzewie ósemkowym.

Kopia pierwsza jest przeznaczona do renderowania. Wierzchołki i indeksy zapamiętane są w buforach w pamięci karty graficznej. Wierzchołki posiadają pełne informacje, łącznie ze współrzędnymi tekstury, wektorem normalnym i wektorami stycznymi.

Drzewo jest słabo rozdrobnione, aby w każdym węźle zawarta była niezbyt mała porcja geometrii. To zapewnia szybsze renderowanie. Fragmenty geometrii pamiętane przez każdy z węzłów pogrupowane są według używanego materiału.

Druga kopia mapy przeznaczona jest do liczenia kolizji. Jej wierzchołki zawierają wyłącznie pozycję (w celu bardziej zaawansowanych obliczeń kolizji należałoby dodać także wektory normalne) i są pamiętane w tablicy w pamięci systemowej. Drzewo ósemkowe jest rozdrobnione, tak aby jak najwięcej trójkątów odrzucać bez dokładnego testowania.

# Rozdział 4

# Przykłady

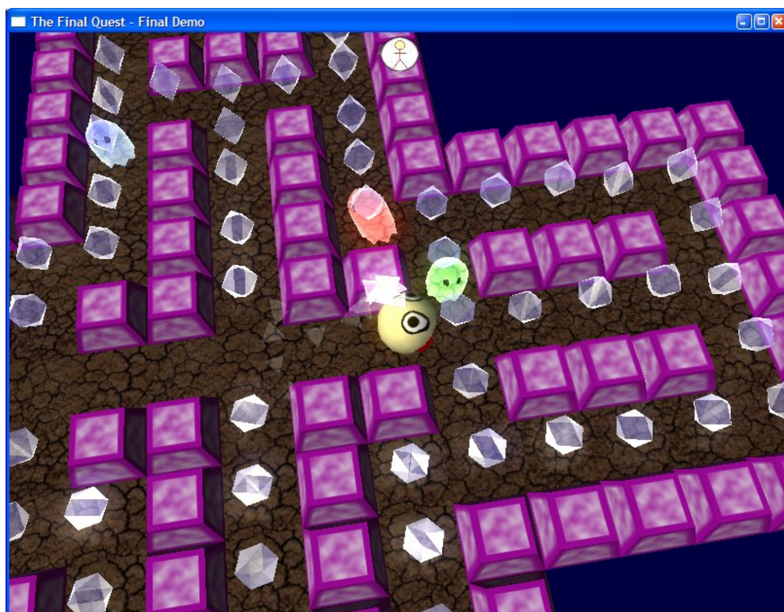
Niniejszy rozdział zawiera kolorowe zrzuty ekranu, które pokazują efekt działania systemu omówionego w tej pracy. Warto podkreślić, że wszystkie przedstawione tu zrzuty ekranu zostały wyrenderowane w czasie rzeczywistym przez program opisany w niniejszej pracy, stworzony przez autora.

## 4.1. Przykłady gier komputerowych

Ponieważ silnik jest biblioteką, konieczne było napisanie programu „klienta", który będzie stanowił namiastkę prawdziwego kodu wykorzystującego taką bibliotekę. Kod zgromadzony w katalogu `Client` stanowią proste prototypy 5 gier różnego gatunku. Zostały one tak pomyślane, aby pokazać wszystkie efekty graficzne oferowane przez silnik, a zarazem w efektowny sposób zaprezentować jego potencjalne zastosowania. Są to:

`GamePacman` — gra typu *Pacman*. Grafika jest prosta, kolorowa. Cienie są wyłączone. Prezentuje możliwości materiałów na przykładzie błyszczących kryształków i półprzezroczystych duchów, a także efekty cząsteczkowe. Patrz rys. 4.1.

`GameRts` to gra typu RTS (ang. *Real-Time Strategy* — strategia czasu rzeczywistego). Prezentuje teren z drzewami, trawą i wodą w widoku od góry. Modele rycerzy używają mechanizmu *Team Color*, aby zmieniać kolor niektórych części tekstury zależnie od drużyny do której należą, a także podlegają animacji szkieletowej. Patrz rys. 4.2.




Rys. 4.1. Przykładowa gra nr 1 typu *Pacman*.



Rys. 4.2. Przykładowa gra nr 2 typu RTS (ang. *Real–Time Strategy* — strategia czasu rzeczywistego).

`GameSpace` to gra polegająca na sterowaniu statkiem kosmicznym latającym nad powierzchnią obcej planety. Pokazana jest w niej przestrzeń otwarta, wykorzystana do renderowania krajobrazu pozaziemskiego. Teren stanowią brązowe skały i kratery, a na niebie widoczne są brązowe chmury i odległa planeta. Broń stosowana przez statek gracza i przeciwnika wykorzystuje efekty cząsteczkowe i promienie (wstęgi). Możliwość celowania prezentuje obliczanie kolizji promienia z obiektami sceny oferowane przez silnik. Ponadto gra wykorzystuje efekt sprzężenia zwrotnego do zasymulowania prostego rozmycia ruchu (ang. *Motion Blur*). Sztuczna inteligencja statku przeciwnika oparta została na logice rozmytej. Patrz rys. 4.3.



Rys. 4.3. Przykładowa gra nr 3, polegająca na sterowaniu statkiem kosmicznym latającym nad powierzchnią obcej planety.

`GameFpp` to gra typu FPS (ang. *First Person Shooter* — strzelanka z perspektywy pierwszej osoby). Jako jedyna z przykładowych gier wykorzystuje jako otocznie mapę zamkniętą wczytaną z pliku QMAP. Ważną rolę odgrywają w niej dynamiczne oświetlenie i cienie, a także nakładany na powierzchnie *Normal Mapping*. Patrz rys. 4.4.



Rys. 4.4. Przykładowa gra nr 4 typu FPS (ang. *First Person Shooter* — strzelanka z perspektywy pierwszej osoby.

`GameRpg` to najbardziej zaawansowana z gier. Ma w założeniu symulować środowisko gier typu MMORPG (ang. *Massively Multiplayer Online Role-Playing Game*). Środowisko gry stanowi teren otwarty o dużej powierzchni, widziany z perspektywy trzeciej osoby (TPP – ang. *Third Person Perspective*). Model bohaterki jest animowany, a do przełączania animacji stania i chodzenia pod wpływem sterowania z klawiatury

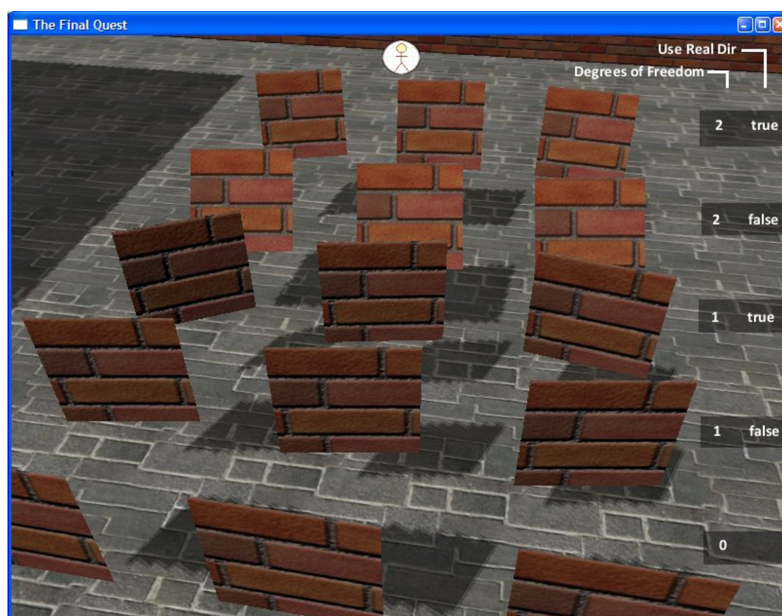
wykorzystywane jest płynne przejście. Gra prezentuje wszelkie dostępne efekty przestrzeni otwartych (woda, drzewa, trawa, opady deszczu i śniegu, falowanie roślin na silnym wietrze, niebo z chmurami), a także efekty cząsteczkowe i inne. Patrz rys. 4.5.



Rys. 4.5. Przykładowa gra nr 5, mająca symulować środowiko gier typu MMORPG.

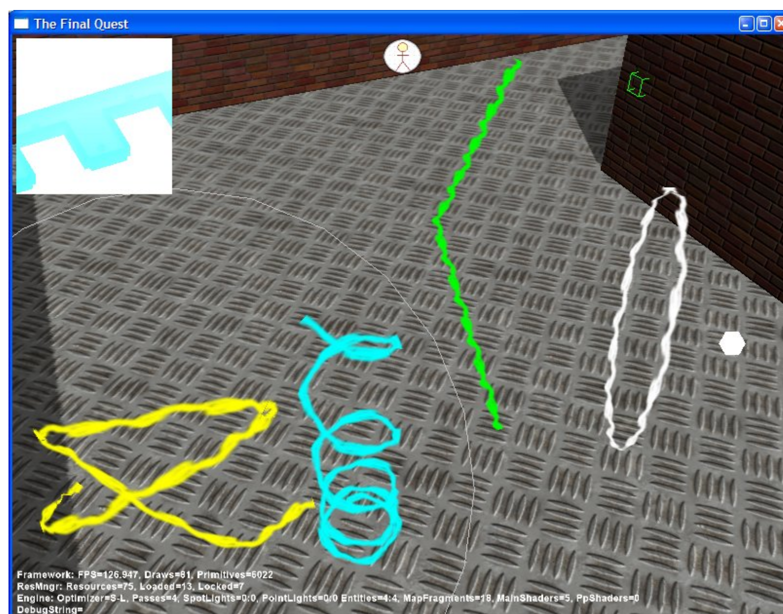
## 4.2. Przykłady efektów graficznych

Rys. 4.6 powstał w celu zilustrowania parametrów sterujących zachowaniem encji typu `QuadEntity`.

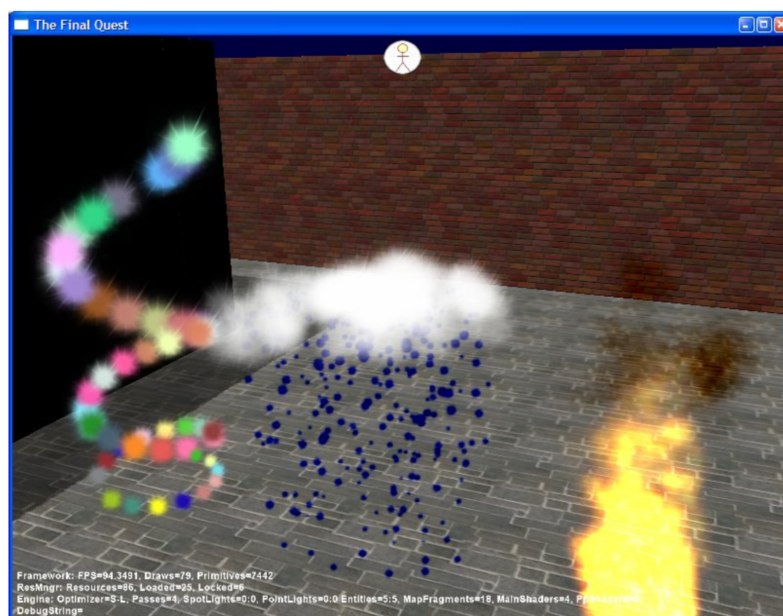



Rys. 4.6. Ułożenie *quadów* zależnie od parametrów obiektu klasy `engine::BillboardEntity`.

Rys. 4.7 zawiera przykłady pokazujące możliwości encji typu `StripeEntity`. Rys. 4.8 pokazuje inny typ encji przydatny do tworzenia efektów specjalnych — efekt cząsteczkowy.

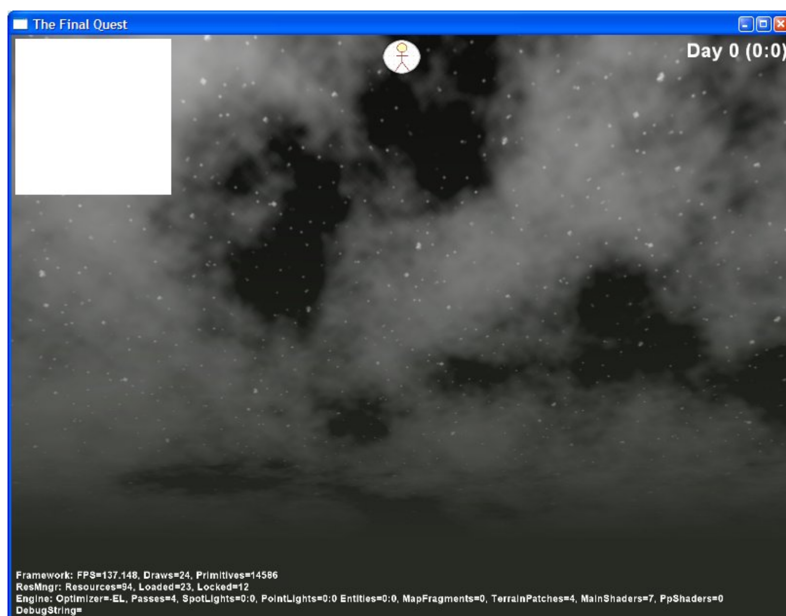



Rys. 4.7. Przykładowy wygląd różnego rodzaju „pasków” typu `engine::StripeEntity`.

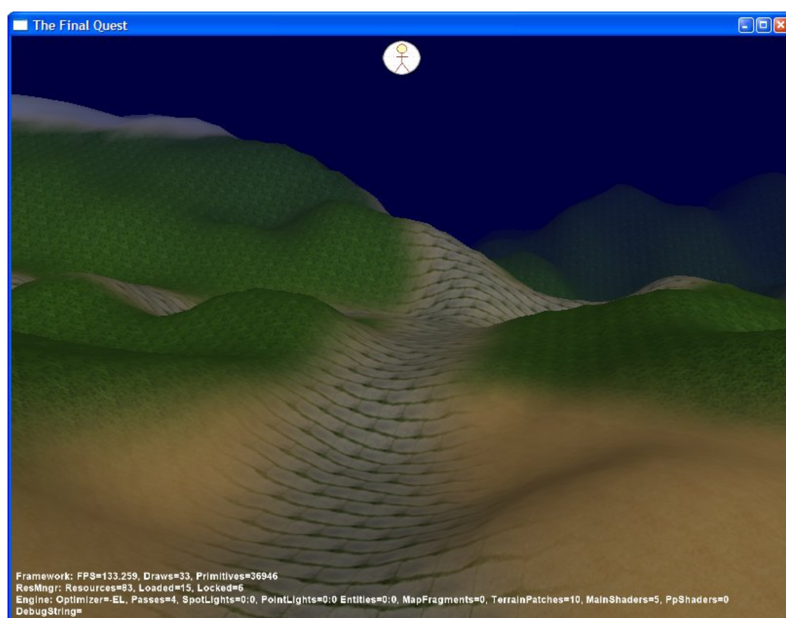



Rys. 4.8. Złożenie 5 encji efektów cząsteczkowych klasy `engine::ParticleEntity` dla zaprezentowania 3 przykładów zastosowania — magiczne cząsteczki, chmura z deszczem oraz ogień i dym.

Rys. 4.9 pokazuje niebo nocą, wyraźnie ilustrując kształt generowanych proceduralnie chmur. Rys. 4.10 pokazuje teren, wyrenderowany dla czytelności bez żadnych dodatkowych obiektów na jego powierzchni.




Rys. 4.9. Niebo nocą jako przykład renderowania proceduralnie generowanych chmur.




Rys. 4.10. Przykładowy teren. Widoczne przejście między formami terenu zależne od wysokości (piasek, trawa, śnieg), jak również wyznaczone przez projektanta (ścieżka).

Rys. 4.11 pokazuje wygląd wody z widocznym odblaskiem od Słońca. Rys. 4.12 pokazuje fragment terenu pokryty drzewami i trawą.




Rys. 4.11. Przykład renderowania wody z widocznym odblaskiem od światła kierunkowego.




Rys. 4.12. Przykład renderowania przestrzeni otwartych. Widoczny teren, niebo, drzewa oraz trawa.

Rys. 4.13 pokazuje efekt opadów atmosferycznych na przykładzie zamieci śnieżnej. Rys. 4.14 pokazuje przykład nietypowego materiału, jaki można uzyskać manipulując parametrami klasy materiału.



Rys. 4.13. Zamieć śnieżna jako przykład renderowania efektów atmosferycznych.




Rys. 4.14. Przykład zastosowania mapowania środowiskowego z użyciem specjalnie przygotowanej tekstury sześciennej wraz z półprzezroczystością i *Fresnel Term* do otrzymania nietypowego materiału.

Rys. 4.15 pokazuje efekt błysku soczewek nakładany w sposób dwuwymiarowy na obraz sceny.




Rys. 4.15. Efekt błysku soczewek pomaga w efektowny sposób przedstawić Słońce.

# Rozdział 5

# Podsumowanie

W ramach pracy stworzony został silnik grafiki trójwymiarowej. Jego kod posłużył jako przykład do opisania architektury i implementacji takiego silnika. Nie może on oczywiście dorównać możliwościami ani jakością silnikom rozwijanym przez wiele lat i przez wielu ludzi, jakie dostępne są na rynku. Stanowi jednak możliwie efektowny, kompletny oraz — co bardzo ważne — ukończony projekt, który może teoretycznie znaleźć zastosowanie przy tworzeniu gier komputerowych i nie tylko.

Kod silnika dołączony do niniejszej pracy na płycie CD był pisany przez jedną osobę (autora) przez około pół roku. Liczy ponad 84 tys. linii. Do jego stworzenia potrzebne były wiedza, doświadczenie i fragmenty kodu wypracowane przez autora w toku wieloletniej nauki programowania. Cały silnik, jak również wersja binarna programu demonstracyjnego, zrzuty ekranu i materiały video udostępnione są również na stronie internetowej autora — `http://regedit.gamedev.pl`. Silnik można pobrać wraz z kodem źródłowym na licencji GNU GPL. Implementacja, jakkolwiek nie wprowadza żadnego poważnego nowego rozwiązania, zawiera wiele nowatorskich pomysłów autora — np. progresywny dysk Poissona (p. 2.3) czy sposób realizacji efektu opadów atmosferycznych (p. 3.8).

Pośród **trudności**, jakie powstały podczas pisania kodu silnika wymienić należy na pierwszym miejscu implementację poszczególnych efektów graficznych. Samo zrozumienie i zamodelowanie danego efektu to jednak dopiero połowa pracy. Nie mniej wysiłku kosztowało zintegrowanie danego efektu z resztą silnika tak, aby stał się na przykład jedną z właściwości materiału czy jednym z efektów postprocessingu. Trudne było zapanowanie nad kodem shadera głównego, który wydłużał się bardzo szybko i ze względu na budowę według modelu subtraktywnego był trudny w poprawianiu i konserwacji. Zorganizowanie procesu renderowania poszczególnych elementów sceny w sposób wydajny trudne było do pogodzenia z chęcią zapewnienia prostoty i elegancji kodu oraz zachowania zasad programowania obiektowego. Użycie własnego formatu siatek — QMSH — wymagało napisania wtyczki eksportującej

do programu graficznego Blender, a do tego z kolei konieczne było opanowanie w dobrym stopniu zarówno języka skryptowego Python, jak i obsługi programu graficznego Blender i API, jakie udostępnia dla wtyczek pisanych w tym języku. Wreszcie, spośród mechanizmów silnika najtrudniejsze okazało się chyba opracowanie algorytmu zapisywania, przeliczania i renderowania siatek korzystających z animacji szkieletowej, a także renderowanie cieni.

**Możliwości rozbudowy** Opisany w niniejszej pracy silnik graficzny został uznany za ukończony. Autor zdaje sobie jednak sprawę z licznych ograniczeń, jakie posiada ta praca i z istnienia technik i efektów, które byłyby pożądane, a które nie zostały zaimplementowane z powodu ograniczeń czasowych i innych.

Pośród efektów graficznych brakuje na przykład **refleksji**, czyli wszelkich odbić, które pozwalałaby na renderowanie luster, powierzchni wody odbijającej krajobraz na horyzoncie czy powierzchni metalicznych. Do przedstawienia materiałów takich jak cegły przydałaby się technika **mapowania nierówności** bardziej zaawansowana, niż zaimplementowany tu *Normal Mapping* — np. *Parallax Mapping* czy *Cone Step Mapping*.

Pewnym ograniczeniem jest oddzielne traktowanie niektórych rodzajów obiektów, np. terenu, drzew, trawy, które uniemożliwia zastosowanie do nich pełnej gamy dostępnych efektów materiału i oświetlenia. Przydałby się na przykład *Normal Mapping* terenu i kory drzew albo rzucanie i otrzymywanie cienia przez trawę. Dużym ograniczeniem może też być fakt, że materiały półprzezroczyste są zupełnie osobną kategorią materiałów i nie podlegają oświetleniu, ponieważ z tego powodu normalne obiekty nie mogą płynnie pojawiać się i znikać.

Zastosowanie **LOD** do różnych elementów silnika znacznie poprawiłoby wydajność lub pozwoliłoby na renderowanie większej liczby obiektów. Obecnie poziom szczegółowości jest dynamicznie wybierany jedynie dla terenu, efektów cząsteczkowych i trawy. Możnaby zastosować tą technikę także do siatek modeli, do drzew, a nawet do materiałów (powierzchnie oddalone od kamery mogłyby być renderowane bez kosztownych obliczeniowo efektów). Do przełączania poziomów szczegółowości siatek — w tym modeli i terenu — powinna być zastosowana pewna forma **CLOD**, aby nie było widoczne „przeskakiwanie" między poszczególnymi poziomami.

Uniwersalny silnik powinien być wyposażony w różne metody obliczania oświetlenia i cieni. Opisany tutaj kod posiada wyłącznie oświetlenie dynamiczne i dynamiczne cienie renderowane metodą *Shadow Mapping*. Przydatne mogłoby być też **oświetlenie statyczne** (*Lightmapping* czy *Radiosity Normal Mapping*), a także cienie renderowane drugą popularną metodą — ***Shadow Volume***. Sam *Shadow Mapping*, w przypadku świateł kierunkowych i scen typu otwartego, powinien być wyposażony w rodzaj **reparametryzacji** poprawiających jakość cieni — np. LiSPSM, TSM lub XSM.

Poprawić wydajność mógłby także inny rodzaj podziału przestrzeni. Zastosowane

w silniku swobodne drzewa ósemkowe są dobre i uniwersalne, ale w scenach typu zamkniętego lepiej sprawdziłyby się **portale**. W kwestii wydajności, sama organizacja procesu renderowania sceny również dałaby się pod pewnymi względami ulepszyć. Na przykład renderowanie mogłoby przyspieszać lepsze zarządzanie stanami urządzenia Direct3D i stałymi przekazywanymi do shadera, tak aby nie były one wszystkie ustawiane przed renderowaniem każdej encji.

Manager zasobów mógłby zostać przepisany w taki sposób, aby zasoby mogły być wczytywane **asynchronicznie**, w osobnym wątku. Wczytywanie danych z plików możnaby uogólnić na pobieranie ich z dowolnego **strumienia danych**, co pozwoliłoby na przechowywanie danych programu w postaci spakowanej w VFS (ang. *Virtual File System*) czy też ich pobieranie prosto z Internetu. Obiekty zajmujących dużo pamięci, jak mapa czy teren, mogłyby być **strumieniowane** przechowując w pamięci tylko wybrane fragmenty. Parametry wielu obiektów, np. mapa wysokości terenu, powinny móc zmieniać się **dynamicznie** podczas działania silnika bez konieczności ponownego tworzenia całego obiektu i wczytywania wszystkich danych z plików na dysku. W przypadku, gdyby w oparciu o ten silnik powstawał edytor, byłoby to koniecznością.

W kwestii architektury trzeba przyznać, że silnik jest słabo rozszerzalny i uogólniony. Trudno byłoby dopisać do niego nowy efekt materiałowy czy efekt postprocessingu. Częściowo wynika to założeń, zgodnie z którymi silnik ten ma być biblioteką zamkniętą, ukrywającą szczegóły implementacyjne. Jednak organizację kodu możnaby poprawić w wielu miejscach, na przykład budując shader główny w sposób addytywny, a nie subtraktywny.

Silnik jako biblioteka, aby mieć większą wartość w oczach potencjalnych użytkowników, mógłby zostać rozbudowany o dodatkowe narzędzia i inne materiały. Oprócz eksporterów do programu graficznego Blender przydałyby się **eksportery** do innych popularnych programów, np. 3ds Max, Maya, a także **konwertery** z różnych formatów modeli do formatu QMSH. Sam kod silnika wartoby skompilować do postaci pojedynczej **biblioteki LIB lub DLL**, tak aby był gotowy do użycia w programie. Wreszcie, aby zastosować go w poważnym projekcie, potrzebna byłaby dobra **dokumentacja** jego API, a także dodatkowe programy narzędziowe z graficznym interfejsem użytkownika (przede wszystkim **edytory**), zastępujące konieczność edytowania plików tekstowych w specjalnych formatach oraz tekstur narzędziowych przedstawiających na swoich pikselach np. rozmieszczenie drzew i trawy. Wypada też jeszcze raz powtórzyć, że opisany w tej pracy kod to jedynie silnik graficzny (renderer). Czym innym jest tzw. silnik gry, który oprócz możliwości graficznych implementuje także dźwięk, fizykę, sztuczną inteligencję i inne potrzebne funkcje.

# Bibliografia

[1] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, R. L. Phillips, *Wprowadzenie do grafiki komputerowej*, Wydawnictwa Naukowo–Techniczne, 2001.

[2] J. Grębosz, *Symfonia C++*, Oficyna Kallimach, 1999

[3] M. McCuskey, *Programowanie gier w DirectX*, Mikom, 2003.

[4] R.S. Wright jr, M. Sweet, *OpenGL. Księga eksperta*, Helion, 1999

[5] F. Dunn, I. Parberry, *3D Math Primer for Graphics and Game Development*, Wordware Publishing, 2002.

[6] W. Jawor, *Principia Silnika*, [b.m.r. i w.].

[7] S. Zerbst, O. Düvel, *3D Game Engine Programming*, Course Technology PTR, 2004.

[8] A. LaMothe, *Triki najlepszych programistów gier 3D. Vademecum profesjonalisty*, Helion, 2004.

[9] A.S. Winter, *An Investigation into Real–Time 3D Polygon Rendering Using BSP Trees*, 1999, `http://citeseer.ist.psu.edu/winter99investigation.html`.

[10] R. Finkel, J.L. Bentley, *Quad Trees: A Data Structure for Retrieval on Composite Keys*, Acta Informatica 4 (1): 1974, 1–9.

[11] T. Ulrich, *Loose Octrees*, w: M. DeLoura, *Game Programming Gems 1*, Charles River Media, 2000.

[12] G. McTaggart, *Half–Life 2 / Valve Source Shading*, Direct3D Tutorial Day, Game Developer's Conference, 2004.

[13] H. Gouraud, *Continuous shading of curved surfaces*, IEEE Transactions on Computers, 20(6): 1971, 623–628.

[14] B. Tuong Phong, *Illumination of Computer–Generated Images*, Department of Computer Science, University of Utah, UTEC–CSs–73–129, 1973.

[15] J. F. Blinn, *Models of light reflection for computer synthesized pictures*, Proc. 4th annual conference on computer graphics and interactive techniques, 1977.

[16] T. Porter, T. Duff, *Compositing Digital Images, Computer Graphics*, 18(3), 1984, 253–259.

[17] J. F. Blinn, M. E. Newell, *Texture and reflection in computer generated images*, Communications of the ACM, 19(10), 1976, 542–547.

[18] J. Mitchell, G. McTaggart, C. Green, *Shading in Valve's Source Engine*, SIGGRAPH, 2006.

[19] M. J. Kilgard, *Hardware Accelerated Anisotropic Lighting*, Game Developers Conference, 1999.

[20] J. Isidoro, C. Brennan, *Per–pixel Strand Based Anisotropic Lighting*, w: W. Engel, *ShaderX*, Wordware Publishing, 2002.

[21] E. E. Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Ph.D. thesis, Dept. of CS, U. of Utah, 1974.

[22] J. F. Blinn, *Simulation of Wrinkled Surfaces*, Computer Graphics, Vol. 12 (3), pp. 286–292 SIGGRAPH–ACM, 1978.

[23] T. Kaneko i in., *Detailed Shape Representation with Parallax Mapping*, w: Proceedings of ICAT 2001, pp. 205–208.

[24] F. Policarpo, M. M. Oliveira, J. o Comba, *Real-Time Relief Mapping on Arbitrary Polygonal Surfaces*, ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games, Washington, DC, April 3–6, 2005, pp. 155–162, `http://www.inf.ufrgs.br/~oliveira/RTM.html`.

[25] N. Tatarchuk, *Practical Dynamic Parallax Occlusion Mapping*, SIGGRAPH 2005.

[26] J. Dummer, *Cone Step Mapping: An Iterative Ray–Heightfield Intersection Algorithm*, `http://www.lonesock.net/files/ConeStepMapping.pdf`.

[27] F. Policarpo, M. M. Oliveira, *Relaxed Cone Stepping for Relief Mapping*, w: H. Nguyen, *GPU Gems 3*, Addison–Wesley, 2007.

[28] F. C. Crow, *Shadow algorithms for computer graphics*, w: J. George *In Computer Graphics*, (SIGGRAPH '77 Proceedings), 1977), vol. 11, pp. 242–248.

[29] L. Williams, *Casting curved shadows on curved surfaces. In Computer Graphics*, SIGGRAPH '78 Proceedings, 1978, vol. 12, pp. 270–274.

[30] W. T. Reeves, D. H. Salesin, R. L. Cool, *Rendering antialiased shadows with depth maps. In Computer Graphics*, SIGGRAPH '87 Proceedings, 1987, Stone M. C., (Ed.), vol. 21, pp. 283–291.

[31] C. Everitt, A. Rege, C. Cebenoyan, *Hardware Shadow Mapping*, nVidia Corporation, `http://developer.nvidia.com/object/hwshadowmap_paper.html`.

[32] S. Brabec, T. Annen, H.P. Seidel, *Shadow Mapping for Hemispherical and Omnidirectional Light Sources*, Computer Graphics Group, Max–Planck–Institut für Infomatik.

[33] M. Stamminger, G. Dreitakis, *Perspective shadow maps*, w: Siggraph 2002 Conference Proceedings (2002), vol. 21, 3, pp. 557–562.

[34] M. Wimmer, D. Scherzer, W. Purgathofer, *Light Space Perspective Shadow Maps*, Vienna University of Technology, Austria, Eurographics Symposium on Rendering (2004), revised version June 10, 2005.

[35] T. Martin, T.–S. Tan, *Anti–aliasing and Continuity with Trapezoidal Shadow Maps*, School of Computing, National University of Singapore, Eurographics Symposium on Rendering (2004).

[36] V. Gusev, *Extended Perspective Shadow Maps (XPSM)*, `http://xpsm.org/`.

[37] Virtual Terrain Project, `http://www.vterrain.org/`.

[38] T. Polack, *3D Terrain Programming*, Premier Press, 2003.

[39] W. H. de Boer, *Fast Terrain Rendering Using Geometrical MipMapping*, 2000.

[40] H. Vistnes, *GPU Terrain Rendering*, w: M. Dickheiser, *Game Programming Gems 6*, Charles River Media, 2006.

[41] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, M.B. Mineev–Weinstein, *ROAMing Terrain: Real–time Optimally Adapting Meshes*, `http://www.cognigraph.com/ROAM_homepage/`.

[42] F. Losasso, H. Hoppe, *Geometry clipmaps: Terrain rendering using nested regular grids*, Siggraph 2004.

[43] C. Bloom, *Terrain Texture Compositing by Blending in the Frame–Buffer (aka "Splatting" Textures)*, 2000, `http://www.cbloom.com/3d/techdocs/splatting.txt`.

[44] J. Weber, J. Penn, *Creation and Rendering of Realistic Trees*, `http://portal.acm.org/citation.cfm?id=218427`.

[45] SpeedTree, `http://www.speedtree.com/`.

[46] K. Pelzer, *Rendering Countless Blades of Waving Grass*, w: R. Fernando, *GPU Gems*, Addison–Wesley, 2004.

[47] D. Whatley, *Toward Photorealism in Virtual Botany*, w: M. Pharr, *GPU Gems 2*, Addison–Wesley, 2005.

[48] K. Boulanger, S. Pattanaik, K. Bouatouch, *Rendering Grass in Real–Time with Dynamic Light Sources and Shadows*, INRIA, 2006, `http://hal.inria.fr/docs/00/08/84/12/PDF/RR--5960.pdf`.

[49] K. Perlin, *Making Noise*, GDCHardCore gamers workshop, San Francisco, Dec 9, 1999, `http://www.noisemachine.com/talk1/`.

[50] Wang Niniane, *Realistic and Fast Cloud Rendering*, 2003, `http://ofb.net/~niniane/clouds-jgt.pdf`.

[51] N. Tatarchuk, *Artist-Directable Real–Time Rain Rendering in City Environments*, SIGGRAPH 2006.

[52] M. Finch, C. Worlds, *Effective Water Simulation from Physical Models*, w: R. Fernando, *GPU Gems*, Addison–Wesley, 2004.

[53] E. Reinhard, G. Ward, P. Sumanta, P. Debevec, *High Dynamic Range Imaging: Acquisition, Display, and Image–Based Lighting*, Westport, Connecticut: Morgan Kaufmann, 2005.

[54] C. Wenzel, *Far Cry and DirectX*, Game Developers Conference 2005.

[55] C. Oat, *Real-Time 3D Scene Post–processing*, ATI Research, `http://ati.amd.com/developer/gdc/GDC2003_ScenePostprocessing.pdf`.

[56] C. Oat, N. Tatarchuk, *Heat and Haze Post–Processing Effects*, w: A. Kirmse, *Game Programming Gems 4*, Charles River Media, 2004.

[57] M. Mocarski, *Zaawansowane i efektywne systemy cząsteczkowe w grach komputerowych*, IV ogólnopolska konferencja inżynierii gier komputerowych, Siedlce 2007.

[58] A. Alexandrescu, *Nowoczesne projektowanie w C++*, WNT, 2005.

[59] T. Lowe, *Critically Damped Ease–In/Ease–Out Smoothing*, w: A. Kirmse, *Game Programming Gems 4*, Charles River Media, 2004.

[60] P. Glinker, *Fight Memory Fragmentation with Templated Freelists*, w: A. Kirmse, *Game Programming Gems 4*, Charles River Media, 2004.

[61] F. P. Placeres, *Improved Frustum Culling*, w: K. Pallister, *Game Programming Gems 5*, Charles River Media, 2005.

[62] R. Burns, M. Sheppard, *Collision Detection and Response*, http://www.harveycartel.org/metanet/tutorials/tutorialA.html.

[63] S. Rabin, *Poster Quality Screenshots*, w: A. Kirmse, *Game Programming Gems 4*, Charles River Media, 2004.

[64] R. H. Carver, Kuo–Chung Tai, *Modern Multithreading*, Wiley–Interscience, 2005.

[65] D. C. Schmidt, I. Pyarali, *Strategies for Implementing POSIX Condition Variables on Win32*, http://www.cs.wustl.edu/~schmidt/win32-cv-1.html.

[66] D. Ohlerich, *Input Lag*, http://www.xyzw.de/c120.html.

[67] R. Fernando, M.J. Kilgard, *Język Cg. Programowanie grafiki w czasie rzeczywistym*, Helion, 2003

[68] D. Filion, *Recombinant Shaders*, w: K. Pallister, *Game Programming Gems 5*, Charles River Media, 2005.

[69] S. Hargreaves, *Generating Shaders From HLSL Fragments*, http://www.talula.demon.co.uk/hlsl_fragments/hlsl_fragments.html.

[70] *Issues with alpha values in texture–mapped billboards*, Virtual Terrain Project, http://www.vterrain.org/Plants/Alpha/.